ECE 457A Lecture Notes



# Artificial Intelligence

## Rational Systems

* **Rational thinking**
  + Uses logical inference to achieve goals
  + Hard to represent informal knowledge
  + Not all problems can be solved like this
* **Rational behaviour**
  + Perceives “world” – given a set of beliefs, acts to achieve goals
  + Controlled by computation
  + More general than inferencing, but can use it
  + May not take “correct” actions, but achieves task at hand

## Swarm Intelligence

* AI technique based on collective behaviour in decentralized, self-organized systems
* Made up of agents who interact with each other and the environment
  + Many individuals
  + Individuals are homogenous
  + Local interactions based on simple rules
  + Self-organization

### Intelligent Agents

* Senses environment and acts on it
* **Rational agent**: for each perceived sequence of events, it does what is expected to maximize performance, based on history and built-in knowledge
* Types of agents:
  + **Simple Reflex Agent**: table-lookup; needs fully observable environment
  + **Model-Based Reflex Agent**: has state information, can handle partially observable environments
  + **Goal-Based Agent**: has goals that help augment knowledge and choose best actions
  + **Utility-Based Agent**: has utility to decide between “good” and “bad” conflicting goals
  + **Learning Agent**: has ability to learn situations and decide how to change to improve performance

## Environments

* Agents work in an environment; certain dimensions of an environment can influence agent design
* **Fully vs. partially observable**
  + **Fully**: sensors detect all aspects of environment relevant to the choice of action
  + **Partially**: noisy/inaccurate sensors, parts of state missing from sensor data
* **Deterministic vs. stochastic**
  + Whether or not the next environment state is completely determined by current state and action of agent
* **Episodic vs. sequential**
  + **Episodic**: choice of action in each episode only depends on episode itself
  + **Sequential**: current choices can affect all future choices – agent must think ahead
* **Static vs. dynamic**
  + Whether or not the environment can change while an agent is deliberating
* **Discrete vs. continuous**
  + Applies to the environment state, how time is handled, and agent perceptions/actions
* **Single-agent vs. multi-agent**

## Adaptive and Cooperative Algorithms

* **Adaptive algorithms**: adjust to new/different situations, can improve their behaviour, evolve, learn by example/discovery, ability to generalize
* **Cooperative algorithms**: a group that solves a joint problem or performs a common task by sharing the responsibility for reaching the goal

# Real-World Problems

* **Well-structured problems**: existing and desired states are clearly identified, and methods to reach desired state are fairly obvious
* **Ill-structured problems**: goals, actions, and end states are unclear, hence methods to reach desired state cannot be found
  + Solution strategies:
    - Retrieve a solution
    - Start from a guess, then improve on it
    - Search amongst alternatives
    - Search forward, from problem to solution
    - Search backward, from goal to problem situations

## Optimization Problems

* Finding the best solution out of all feasible solutions, based on a set of constraints
* They are found everywhere, and can be NP-hard or NP-complete (very hard to solve)
  + NP-hard problems can be solved by exhaustive search, but the size of the instance grows the running time very large for even fairly small-sized problems
* Optimization algorithms: search methods where the goal is to find a solution to an optimization problem
  + **Exact algorithms**: finds the most optimal solution, with high computational cost
  + **Approximate algorithms**: finds the near-optimal solution, with low computational cost
    - Constructive methods: start from scratch, build solution by adding one component at a time
    - Local search methods: start with initial solution, iteratively replace current solution with better one
* **Heuristics** are a solution strategy that uses trial-and-error to produce acceptable solutions to complex problems in a reasonable amount of time
  + They aim to effectively generate good solutions, not necessarily the most optimal
  + Characteristics:
    - Short runtime
    - Easy to implement
    - Flexible
    - Simple

# Local Search Methods: Goal and Problem Formation

## Problem Solving by Searching

* **Search**: moving from state to state in the problem space, look for sequence of actions that lead to a goal (or terminate without finding one)
  + Many optimization problems can be solved using search algorithms
* **Goal-based search agents**: use information about environment and current state to reach end goal
  + Requires information that describes the state of the world, describes available transitions, and what solves the problem
* **Utility-based search agents**: consider the performance of accomplishing the end goal

## Goal and Problem Formulation

### Goal Formulation

* Decide what aspects we are interested in vs. what aspects can be ignored
  + **Goals** limit the search and allow termination
  + Define a goal: what is the agent searching for?
    - Goal test: what it means to achieve the goal
  + Define the **solution**: the goal itself? The path to get to the goal?

### Problem Formulation

* Decide how to manipulate important aspects and ignore the others
  + Represent the search space by **states**
  + Define **actions** that are valid for a given state
    - Actions that an agent can perform, and their cost
    - A transition model
    - Neighbourhood: a set of states that a state S can move to after any action is taken
  + Define the **cost** of actions
* A well-defined problem space includes:
  + **State space**: a partial or complete configuration of a problem
  + **Initial state**: search begins here
  + **Goal state + goal test**: search terminates here
  + **Actions**: defines movement between states
  + **Cost (action and path cost)**: assigns each solution a cost

### Closed World Assumption

* Assume that all the necessary information about a problem is available in each percept, i.e. each state is a **complete description** of the world – no incomplete information at any state
* The environment is:
  + Fully observable
  + Deterministic
  + Sequential
  + Static
  + Discrete

Example: 8-tile puzzle

* **State**: encoded location of 8 tiles and blank
* **Initial state**: any arrangement of tiles
* **Goal state**: predefined arrangement of tiles
* **Actions**: slide blank tile up, down, left, or right, without moving off the grid
* **Cost**: cost of each action (move) is 1; path cost is the total number of moves

Example: Travelling salesman problem (finding shortest route that visits each city once)

* **State**: path connecting cities
* **Initial state**: start at one city, no other cities visited
* **Goal state**: all cities have been visited
* **Actions**: move to an unvisited city
* **Cost**: cost of each action is the distance between cities; path cost is the total distance travelled

Example: Missionaries and cannibals problem

Three missionaries and three cannibals wish to cross the river. They have a small boat that will carry up to two people. If, at any time, cannibals outnumber missionaries on either side of the river, they will eat the missionaries. Find the smallest number of crossings that will allow everyone to cross the river safely.

* **State**: configuration of people on either side of the river and on the boat
* **Initial state**: all people on left side of river
* **Goal state**: all people on right side of river, with no one eaten
* **Actions**: move boat, containing 1 or 2 people, to other side of river and back
* **Cost**: cost of each action is 1; path cost is the total number of crossings

# Local Search Methods: Technologies and Fundamentals

## Search Trees

* Search trees are superimposed on the graph representation of a problem
* While the graph might be finite, the search tree can either be finite or infinite
  + It is infinite when repeated states are allowed – can get into cycles
* Terminology:
  + **Branching factor** : maximum number of children per node
  + **Maximum depth** : finite trees have a maximum depth
  + **Level :** everynodein the search tree exists at a level

## Properties of Search Algorithms

* **Completeness**: is the algorithm guaranteed to find a goal node, if one exists?
* **Optimality**: is the algorithm guaranteed to find the best goal node (lowest path cost)?
* **Time complexity**: number of nodes generated
* **Space complexity**: maximum number of nodes stored in memory

# Local Search Methods: Types and Strategies

* Generic searches are a repetition of **choose**, **test**, and **expand**
* Each search strategy influences how to choose the next node to consider
* Queue: store nodes on the fringe to be expanded

### Uninformed Search

* No information about the likely direction of goal nodes – “blind search”
* Only has information provided by problem formulation
* Examples: breadth-first search, depth-first search, depth-limited search, uniform-cost search

### Informed Search

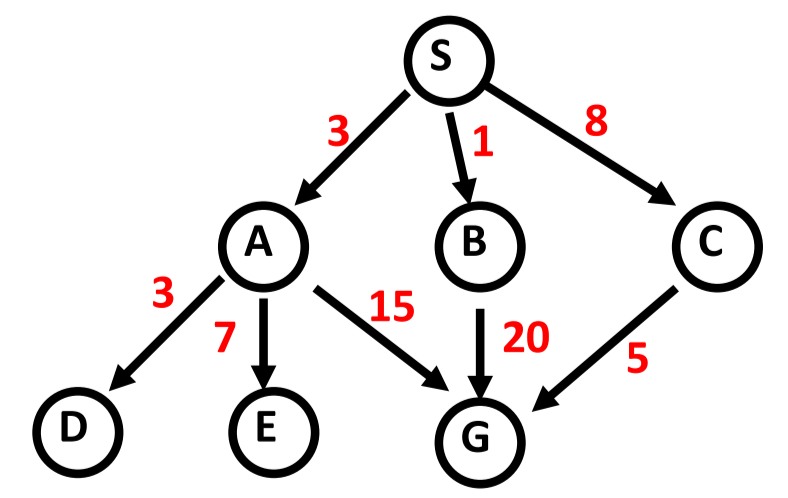
* Uses information about domain to head in general direction of goal nodes – “heuristic search”
* Has additional information to help it judge the estimated cost of a state to the goal
* Examples: hill climbing, best-first search, beam search, A, A\*

# Uninformed Search Strategies

## Breadth-First Search

* Expand shallowest unexpanded node first
  + **Fringe**: FIFO queue with nodes waiting to be expanded; new successors go in end of queue
* **Complete search** if branching factor is finite
* **Optimal** if path cost = depth , guaranteed to return shallowest depth
  + Takes long time to find solutions with large depth
* **Time and space complexity**:

Example: Find the path to node G.

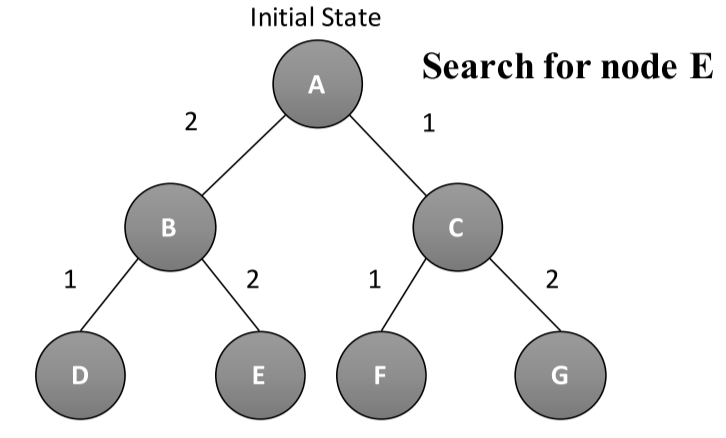


Expanded node| {Queue}

## Uniform-Cost Search

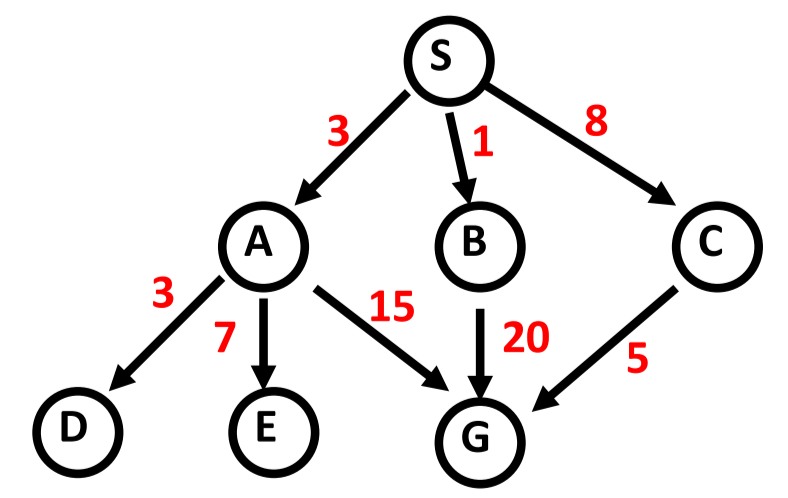
* Expand on the lowest-cost node in the fringe, where cost is the path cost
  + **Closed list**: stores expanded nodes
  + **Open list**: fringe list
* Cannot have any zero-cost or negative-cost edges
* Breadth-first search is uniform-cost search with constant-cost edges
* **Complete** given a finite tree
* **Optimal**
* **Time and space complexity**: , where is the path cost to the goal, and is the minimum cost

Example:



{Open list} | {Closed list}

Example: Find the path to node G.

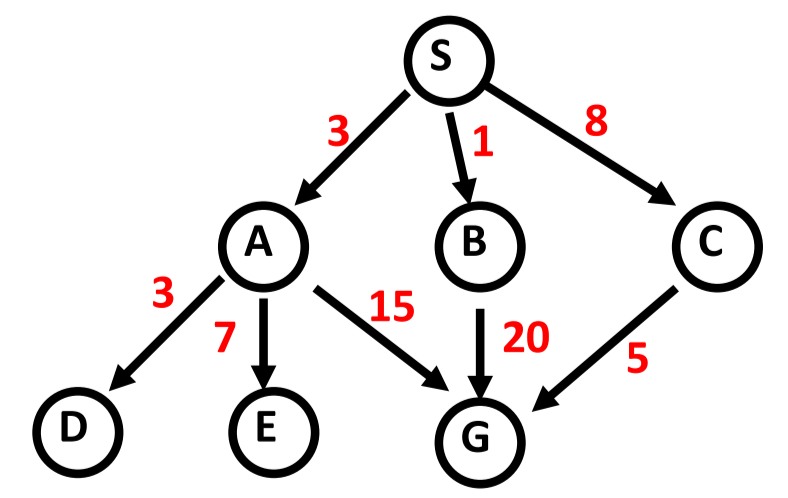


{Open list} | {Closed list}

## Depth-First Search

* Expand the deepest unexpanded node first
  + **Fringe**: LIFO stack, with successors placed in front
* **Complete** if maximum depth of any node is finite; fails for infinite-depth or loops
* **Optimal**: no
* **Time complexity**: – badif m is much larger than , depth of least-cost solution
* **Space complexity**:

Example: Find the path to node G.



Expanded node| {Queue}

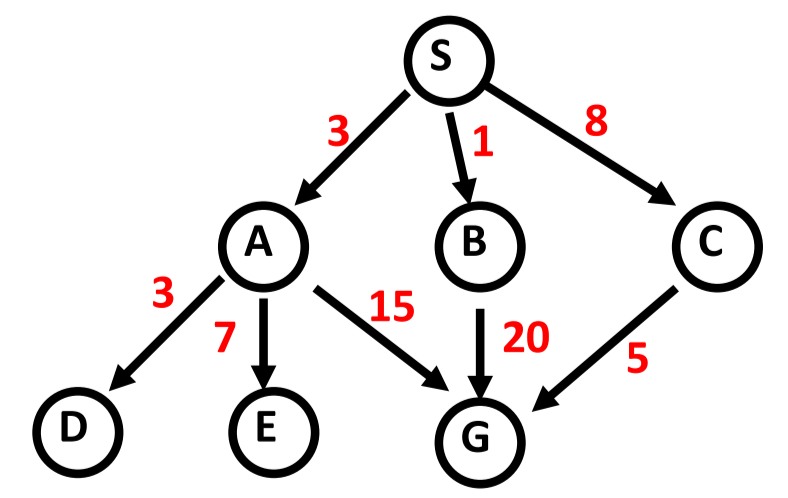
## Depth-Limited Search

* Depth-first search, but with limit on depth of search – prevents algorithm from getting stuck in infinitely deep paths or loops
* **Complete** if there is a solution within depth bound, i.e.
  + If , then the shallowest solution is deeper than the bound, so DLS will fail
  + If , search cost increases compared to breadth-first search
  + Always terminates
* **Optimal**: no
* **Time complexity**:
* **Space complexity**:

## Iterative Deepening Search

* Depth-first search to a fixed depth; if no solution found within depth, then depth is increased and search restarts
* **Complete** if is finite
* **Optimal**: if path cost = depth, returns shallowest goal
* **Time complexity**:
  + Nodes above are generated multiple times
* **Space complexity**:

Example: Find the path to node G.



|  |  |  |
| --- | --- | --- |
| Expanded Node | Queue | Limit |
|  |  | 1 |
|  |  |
|  |  | 2 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  | 3 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Informed Search Strategies

* Search the **most promising** branches of the state-space first, using domain-specific
* Can:
  + Find solution more quickly
  + Find solution in limited time
  + Often finds better solution, since more profitable part of state space can be examined, while ignoring unprofitable parts
* **Heuristic function** : knowledge about the problem used to prune the search, by giving an estimated cost/distance of minimal cost path from to the goal state
  + for all nodes
  + for all goal nodes
  + for all dead-ends that can never lead to a goal
  + A heuristic function is **admissible/optimistic** if it never overestimates the cost of reaching the goal

### Strong vs. Weak Methods

* **Strong methods**: designed to address a specific type of problem
* **Weak methods**: general approaches – can be applied to many types of problems
  + Not tailored to specific situations
  + Do not take advantage of domain-specific heuristics
  + Examples:
    - **Means-ends analysis**: represent current situation and where we want to end up, then look for ways to shrink difference between the two
    - **Space splitting**: list possible solutions to a problem, then rule out classes of possibilities
    - **Subgoaling**: split large problem into several smaller ones that can be solved one at a time

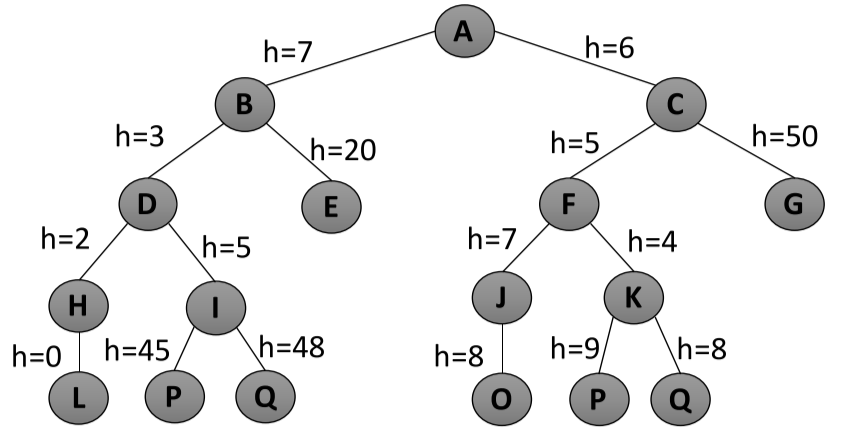
## Best-First Search

* Family of algorithms that select nodes for expansion based on **evaluation function**
* Use a **priority queue**,where nodes are ordered by increasing value of
* “Best”/lowest-value node according to is expanded
  + incorporates domain-specific information

### Greedy Best-First Search

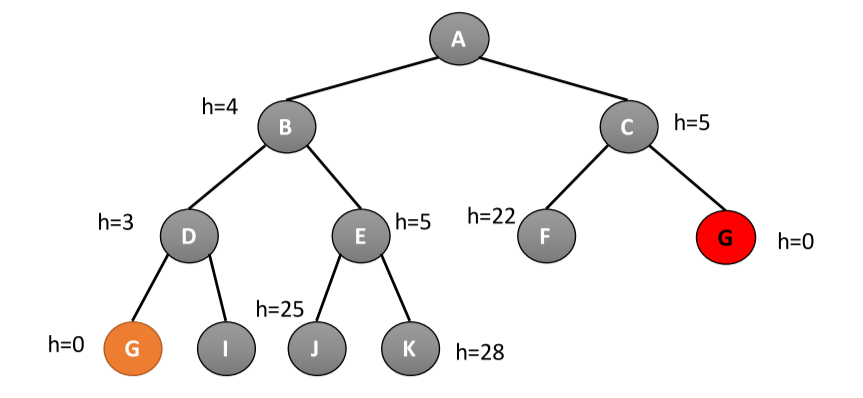
* Uses – selects nodes to expand based on how close it is to goal node
* **Complete**: no, can get stuck in loops
* **Optimal**: no
* **Time/space complexity**: – good heuristic can give dramatic time improvement

Example: Find goal node L.



* Breadth-first search: A, B, C, D, E, F, G, H, I, J, K, L
* Best-first search: A, C, F, K, B, D, H, L

Example: Find goal node G.

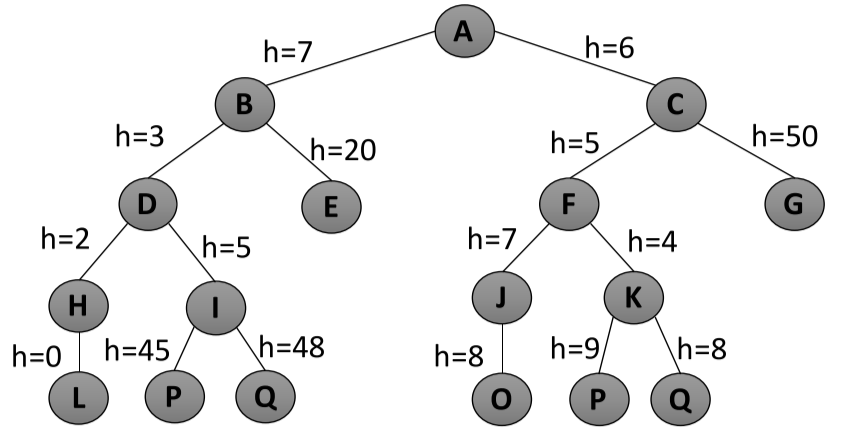


* Breadth-first search: A, B, C, D, E, F, G
* Best-first search: A, B, D, G

## Beam Search

* Informed breadth-first algorithm - aims to minimize memory requirement
* **Beam width** : expands only the first promising nodes at each level (maximum list size)
  + Keeps best nodes as candidates for expansion
* Uses evaluation function
* **Complete**: no
* **Optimal**: no
* **Time/space complexity**:
* **Not admissible**

Example: Find goal node L, with .



## Algorithm A

* Uses evaluation function
  + = **minimal-cost path** from start state to state – adds “breadth-first” component to evaluation function
* Ranks nodes by estimated cost of solution from start node, through node , to goal
* Not complete if can equal
* Not admissible

### Algorithm A\*

* Algorithm A, with constraint that
  + = the true cost of the minimal cost path from state to a goal state
* is admissible when holds, hence A\* is admissible
  + guarantees that the first solution found is an optimal one
* **Complete**: when branching factor is finite and every operator a fixed, positive cost

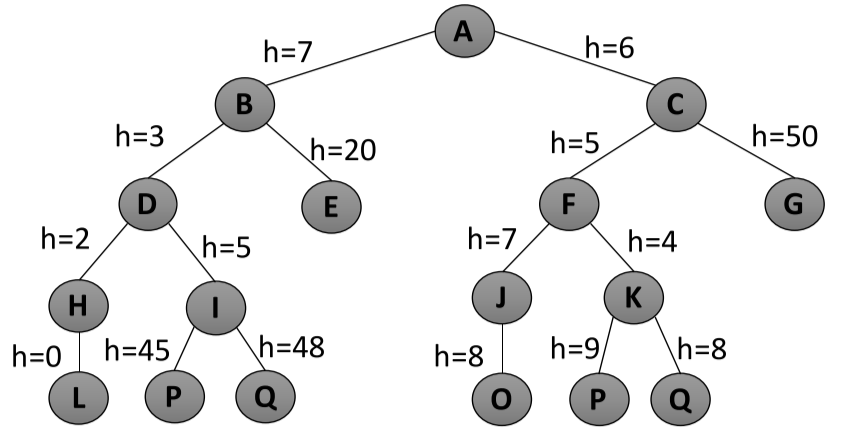
## Types of Heuristics

* **Perfect heuristic**: if for all , then only nodes on the optimal solution path will be expanded, so no extra work will be performed
* **Null heuristic**: if for all , then this is an admissible heuristic and A\* acts like uniform-cost search
* **Better heuristic**: if for all non-goal nodes, then is a better heuristic than
  + If A1\* uses and A2\* uses , then every node expanded by A2\* is also expanded by A1\*
  + Hence, A1\* expands at least as many nodes as A2\*
  + A2\* is better informed than A1\*
* The closer is to , the fewer extra nodes will be expanded
* How to obtain admissible heuristics? 🡪 **relax** the problem rules

## Hill Climbing Search

* Informed depth-first algorithm – aims to improve efficiency
* Iterative algorithm that starts with an arbitrary solution, then tries to find a better solution by incrementally changing a single element of the solution
* Sorts successors of a node according to heuristic values, before adding to list to be expanded
* If there exists a successor for current state that is better than the current state and all successor ( and for all successors of ), move to it; otherwise, halt at
  + Does not allow backtracking or jumping to an alternative path, since it is memoryless
  + Corresponds to beam search with
* **Complete**: no, since search will terminate at local minima or plateaus

Example: Find goal node L.



* A, C, F, K 🡪 does not find L

# Game Playing as Search

* In games with an opponent, need to search for best move, wait for opponent response, then search again for best move
* Limited amount of time to find goal in each search
* In games, objective is not only to find the best way to the goal, but also beat the opponent

## Types of Games

* **Perfect information**: each player has complete information about opponent’s position and available choices (e.g. chess, monopoly)
* **Imperfect information**: each player does not have complete information about opponent’s position and available choices (e.g. poker, bridge)

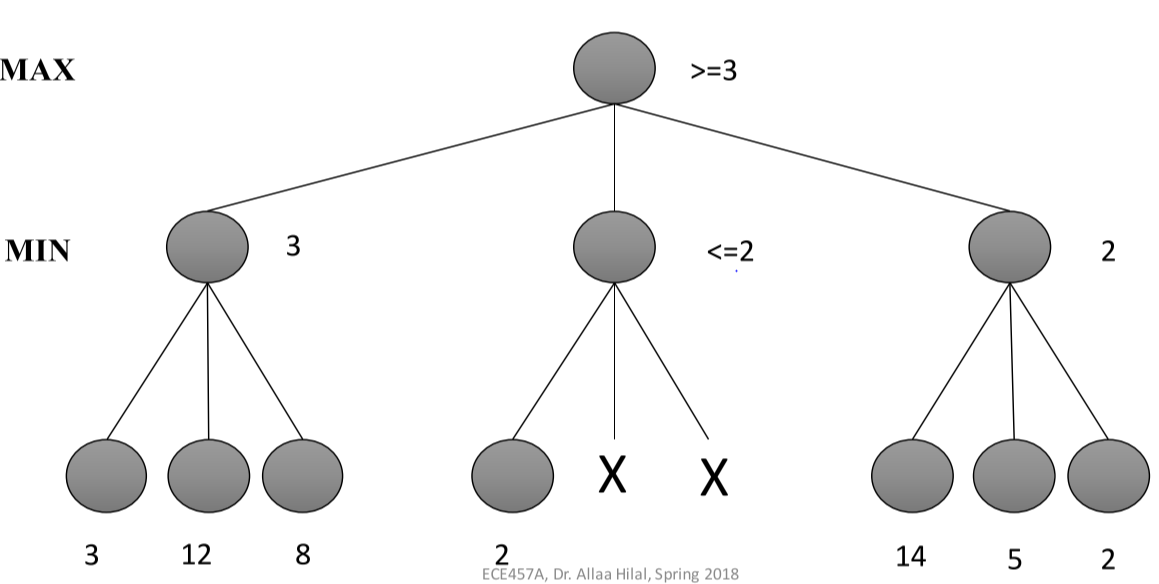
## Max Min Strategy

* In a two-player game with perfect information, ideally, the player expands the game tree and considers all possible opponent moves until end of game, with leaf nodes as win, lose, or draw
* Max min strategy: player – MAX, opponent – MIN
  + Commonly used with **zero sum games**: when one player wins, other loses
* **Minimax principle**: principle for decision-making, where, to pick between two conflicting strategies, use the strategy that minimizes the maximum losses that could occur
* The **minimax value** of a node is the utility of being in the corresponding state, assuming both players play optimally from there to the end of the game
  + Label each level in the game tree with MAX an MIN
  + Label leaves with evaluation of player
  + Go through the game tree:
    - If parent node is a MAX node, label with **maximum value** of its successors
    - If parent node is a MIN node, label with **minimum value** of its successors
* It is only feasible to expand on complete game tree for very simple games; for more complex games, such as chess, it is only feasible to explore a limited depth of game tree
  + **Evaluation function** : Measures the worth of a board configuration to the player
    - The lower in the game tree level, the better the estimate
    - E.g. tic tac toe: number of possible wins not blocked by opponent, minus number of possible wins for opponent not blocked by current player
      * MAX wants to maximize evaluation function
      * MIN wants to minimize evaluation function
* **Complete**: yes, if tree is finite
* **Optimal**: yes, against optimal opponent
* **Time complexity**:
* **Space complexity**: – depth-first exploration

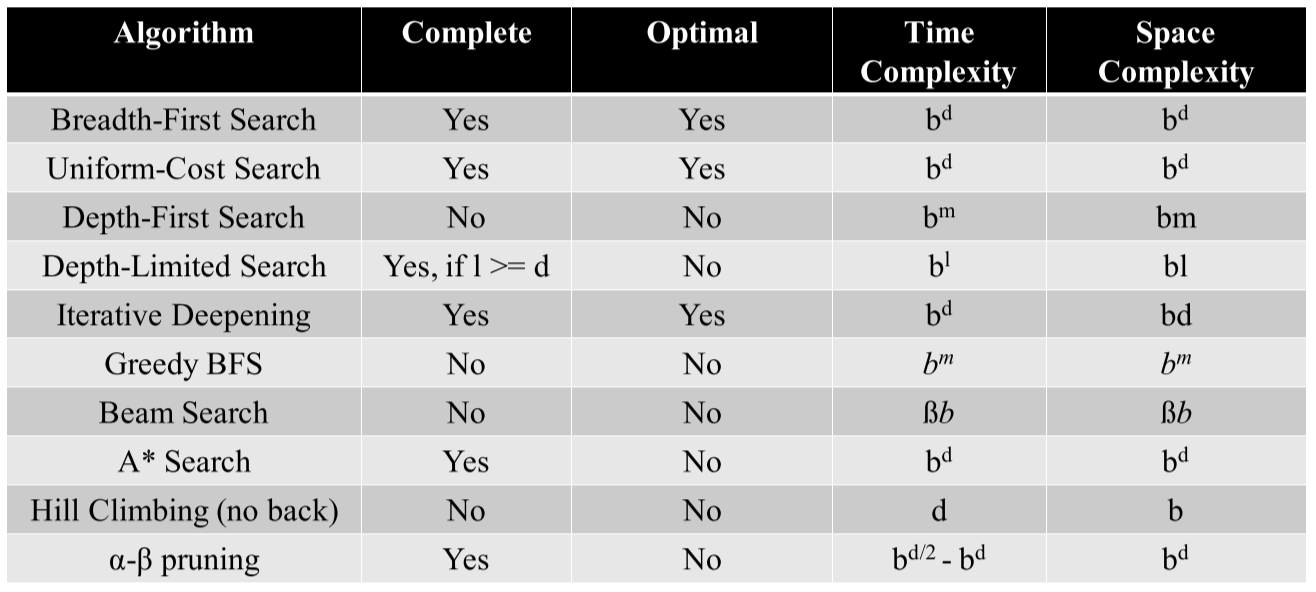
## Alpha-Beta Pruning

* Reduces the number of nodes that need to be generated and evaluated
* Two parameters:
  + : best value of MAX seen so far
  + best value of MIN seen so far
* **Alpha cut-off**: when value of a min position is less than or equal to of parent, stop generating successors (since parent looking for max)
  + Search discontinues below any min node with of one of its ancestors
* **Beta cut-off**: when value of a max position is greater than the of its parent, stop generating successors (since parent looking for min)
  + Search discontinues below any max node with of one of its ancestors
* Returns exactly the same value as Min-Max algorithm
  + Best case, cost reduced from to
  + Worst case, no nodes are pruned

Example:



## Comparing Search Strategies



* branching factor
* depth of optimal solution
* maximum depth
* depth limit
* beam width

# Metaheuristics

* Algorithms that combine heuristics in a higher-level framework, aimed at efficiently and effectively exploring the search space
  + Non-exhaustive search algorithm with internal heuristic
* Strategies that guide the search process
* Goal: effectively explore search space to find (near-) optimal solutions
  + Techniques that constitute metaheuristic algorithms include simple local search procedures to complex learning processes
  + Metaheuristic algorithms are **approximate** and **non-deterministic**
  + They may incorporate mechanisms to avoid getting trapped in confined areas of search space
  + They are not **problem-specific** – basic concepts permit an **abstract level description**
  + They may make use of domain-specific knowledge in the form of heuristics, controlled by **upper level strategy**
* Metaheuristic searches are often hybrid searches:
  + Search identifies neighbourhoods that might be valid locations for search item
  + Search intensifies to find exact right answer
* Types:
  + **Trajectory methods**: handles one solution at a time
    - Uses memory structures to avoid getting stuck in local minima and avoid revisiting visited nodes
  + **Population-based methods**: handles multiple solutions at a time

## Tabu Search

* Combination of local search strategy and memory structures
* **Local search:**
  + Start with feasible solution
  + While terminating criterion not met
    - Generate neighbouring solution by applying series of local modifications/moves
    - If the new solution is better, replace the old one
* **Tabu search**:
  + Forbid/penalize moves that take the solution to points in the solution space previously visited
  + Accept non-improving solutions deterministically, to escape from local optima (where all neighbouring solutions are non-improving)
* **Memory structures**:
  + **Tabu list :** short-term memory based on recency of occurrence – prevents search from revisiting previously visited solutions, or to keep track of good solutions to return to, to intensify search
    - Holds fixed and limited amount of information
    - **Tabu tenure**: the number of iterations , for which we keep a certain move in the list
  + Long-term memorybased on frequency of occurrence – prevents search from visiting frequently-visited solutions, to diversify search
* **Neighbourhood** : nodes to be expanded on
  + To select a new state, consider neighbours not in the tabu list, i.e.
  + can be reduced and modified based on history and knowledge
* **Termination conditions**:
  + No feasible solution in neighbourhood of current solution
  + Reached maximum number of iterations allowed
  + The number of iterations since last improvement is larger than a specified number
  + Evidence shows optimum solution has been obtained
* **Candidate list**: can be used to reduce the number of solutions examined in an iteration
  + Isolates areas of neighbourhood containing desirable moves
* **Aspiration** : allow certain moves even if it is on the tabu list, to prevent stagnation

### Basic Tabu Search Algorithm

* Generate initial solution
* While termination criteria not met
  + Search neighbourhood: check nodes
  + If improves best-known solution , set
  + Update and

### Example of Tabu Restrictions

* A move that involves the same exchange of positions as a tabu move
* A move that involves any of the positions involved in a tabu move

### Tabu Tenure

* **Static**: choose constant
* **Dynamic:** chooserandom between and
* Limitations:
  + Fixed-length tabu lists cannot always prevent cycles, if cycle length is longer than tabu tenure

### Aspiration Criteria

* By default: tabu move is admissible if it yields a solution better than any obtained solution so far
* By objective: tabu move is admissible if it yields a solution better than an aspiration value
* By search direction: tabu move is admissible if direction of search (improving/non-improving) does not change

### Intensification and Diversification

* **Intensification**: exploit small portion of search space, or penalize solutions far from current solution
  + Based on recency memory
  + If certain components always appear in the best solution, without interruption, intensification phase can locally optimize the best-known solution without removing those components
* **Diversification**: forcing search on previously unexplored areas, or penalizing solutions close to current solution
  + Based on frequency memory
  + Tabu search may miss good solutions in unexplored search areas
  + Two major approaches:
    - **Restart diversification**: forces components that rarely appear in current solution, then restart from those points
    - **Continuous diversification**: incorporate component frequency term to evaluation function

## Tabu Search Enhancements

### Adaptation

* Allow the number of iterations a move maintains a tabu status in the tabu list, to vary dynamically
* One approach:
  + Compute tabu tenure range ( and ) in advance based on problem size
  + Chooserandom between and
* Second approach:
  + Tabu tenure restricted between and
  + If current solution is better than best one so far,
  + If in an improving phase,
  + If not in an improving phase,
  + Every iterations, randomly change and

### Cooperation

* **Coordinated searches**: independent, concurrent tabu searches; information exchanged every predetermined number of iterations (synchronous communication)
  + Different search processes use different:
    - Initial solutions
    - Parameter settings
    - Handling incoming solution
      * **Simple import**: always replace its own best-so-far solution
      * **Conditional import**: replace its own best-so-far solution only if incoming solution is better
  + Can reset tabu list after any synchronization
* **Asynchronous communication approach**: each search process broadcasts information when its best-so-far solution is updated
  + Central memory handles information exchange
    - If sent solution is worse than solution in central memory, search process uses stored solution
    - If sent solution is better, replace central memory solution
  + Central memory can also hold pool of best solutions
    - When a search process requests a solution, it gets a randomly-selected one from the pool
* Conclusions:
  + For fixed number of iterations, increasing # processors improved the solution up to a certain point
  + Reducing the number of iterations between synchronization steps increases computation times due to message-passing overhead
  + Conditional import always produces similar/better than simple import

# Simulated Annealing

## Physical Annealing

* Analogy:
  + If a liquid material cools too quickly, the material will solidify into a sub-optimal configuration
  + If the liquid material cools slowly, the crystals in the material will solidify optimally into state of minimum energy (ground state)
* **Ground state**: minimum of cost function in optimization problem
* **Annealing**: finding the minimum of a given function, depending on many variables
  + Physical annealing: thermal process of achieving low-energy state in a solid
    - Heat material until **annealing temperature**
    - **Hold** the material at that temperature until even throughout
    - Cool material **slowly**
  + Ground state of the solid is only obtained if maximum temperature is high enough and cooled slowly

## Simulated Annealing

* Defines a set of conditions that, if met, ensure the random walk will sample from probability distribution at equilibrium
* **Neighbourhood search approach** to move from solution to new solutions; tries to escape from local optimum by allowing non-improving solutions in a controlled manner
* **Acceptance probability:**
  + = change in solution cost
  + = current temperature
* Under constant temperature, process will converge regardless of starting solution, as long as it is possible to find a sequence of exchanges that will transform any solution into another with non-zero probability
* Under non-constant temperature, process will converge if temperature is reduced to zero slowly and the number of iterations at each temperature is large
* **Advantages**: ease of use, provides good solutions for wide range of problems
* **Disadvantages**: large runtime for many runs, many tuneable parameters

### Basic SA Algorithm

* Set current solution to initial solution
* Set current temperature to initial temperature
* Select a **temperature reduction function**
* While stopping condition is not satisfied:
  + While max number of iterations has not been reached for the temperature:
    - Select solution from neighbourhood
    - Calculate change in cost between and
    - If , accept solution:
    - Else:
      * Generate random number in range (0, 1)
      * If , accept solution:
    - Decrease using
* Return

### Strategy

* At high temperatures, explore parameter space – **random walk**
* At lower temperature, restrict exploration – **hill-climbing**
* Always accept better solutions
* Accept worse states according to acceptance probability , which lowers acceptance probability for larger as temperature decreases

### Annealing Schedule

* How the value of temperature is adjusted
* Determined by:
  + **Initial temperature** – must be high enough to allow a move to possibly any state in the search space, but not hot; good initial temperature should accept about **60%** of worst moves
  + **Final temperature** – doesn’t have to reach zero; can stop algorithm at a reasonably low temperature, or when system is frozen (no better moves generated, and no worse moves accepted)
  + **Temperature decrement rule**
    - Linear:
    - Geometric: ,
  + **Number of iterations per temperature** – should allow enough iterations for system to be stable at that temperature
    - Could dynamically change value to allow smaller number of iterations at high temperatures and large number of iterations at low temperatures

## Simulated Annealing Enhancements

### Adaptation

* The most critical parameters in SA are:
  + **Initial temperature** – try different values to see which leads to better solutions
  + **Cooling schedule**
    - ASA automatically adjusts algorithm parameters to control temperature schedule, provided
  + **Number of iterations per temperature**
* **Acceptance probability** – uselookup table to reduce computation time
* **Cost function** – avoid cost functions that return same value for many states, or have dynamically changing weighting of penalty terms

### Cooperation

* **Cooperative SA (COSA)** implements concurrent and synchronous runs of multiple SA processes
  + Algorithm manipulates a **population** of solutions (multiple) at once
  + Concurrent processes are coupled through **cooperative transitions**, which replace uniform distribution used to select neighbours

Example: start with randomly chosen population

* New population is iteratively produced
* Any new solution is cooperatively produced by the previous value of that solution and the previous value of a randomly selected solution  
  E.g.
  + Find neighbours of that are closer to than – these neighbours constitute **closer set**
    - If closer set isn’t empty, randomly select new solution from it
    - Otherwise, randomly select solution from neighbourhood of
* Temperature is updated base on **difference of mean fitness** of new and old populations

# Evolutionary Algorithms

## Biological Evolutionary Inspiration

* Evolutionary analogy:
  + A **population** of individuals exist in an environment with limited resources
  + **Competition** for those resources causes selection of **fitter**, better-adapted individuals
  + These individuals create **new generations** through recombination and mutation
  + New individuals compete for survival (possibly with parents)
  + Over time, **natural selection** cause rise in fitness of population
* Evolutionary algorithms are stochastic, population-based, **“generate and test”** algorithms
  + Recombination and mutation create necessary diversity/novelty
  + Selection reduces diversity, pushes for quality

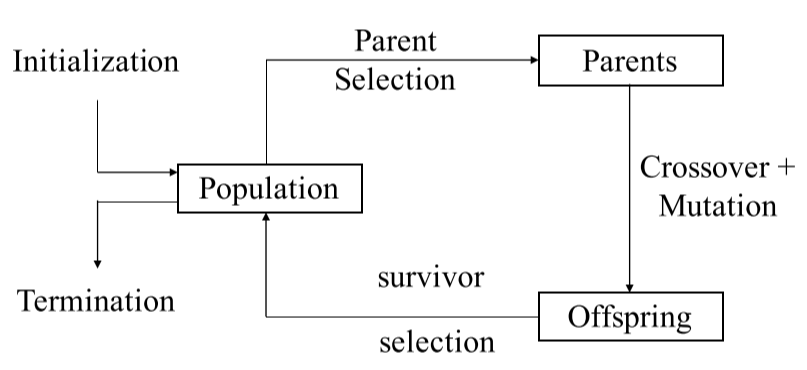
## Active Information

* **Conservation of information theorem**: any search algorithm performs as well as random search, on average, unless it takes advantage of problem-specific information about search target or search space
* Three measures characterize information required for successful search:
  + **Endogenous information**: difficulty of finding a target using random search
  + **Exogenous information**: difficulty in finding a target, once a search takes advantage of problem-specific information
  + **Active information**: difference between endogenous and exogenous information; measures how useful problem-specific information is, to finding a target

# Genetic Algorithms

* Based on Darwinian theory of “**survival of the fittest**” – natural selection favours individuals that compete effectively for resources, and those who adapted to fit environmental conditions best
  + Fitness is determined by **phenotypic** traits behavioural and physical features
  + **Mutations**, small and random variations in phenotypic traits, occur during reproduction
  + Basic operations: **selection**, **reproduction**, **mutation**
* Genetic algorithms maintain **population** of candidate solutions, evolve them by iteratively applying a set of stochastic operators

## Simple Genetic Algorithms (SGA)



* Population of solutions evolves from one iteration to the next, based on selection and search/genetic operators:
  + **Recombination** **(crossover)**: 1-point, N-point, or uniform
  + **Mutation**: bitwise bit-flipping with fixed probability

### SGA Algorithm

* Initialize population with random candidates
* Evaluate all individuals
* While termination criteria not met:
  + Select parents from population
  + Shuffle mating pool
  + Apply crossover
  + Mutate offspring – bit-flip with independent probability for each bit
  + Replace population with current generation
* Possible termination criteria:
  + Specified number of generations
  + Minimum threshold
  + No improvement in best individual for specified number of generations
  + Memory/time constraints

## SGA Characteristics

### Representation

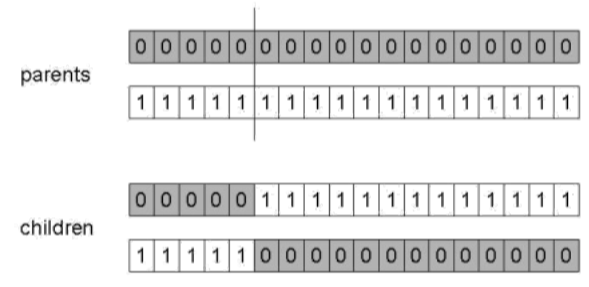
* Each **phenotype** is mapped to a **genotype** (chromosome) – **binary string** (0/1 for presence/absence of a trait)

### Selection

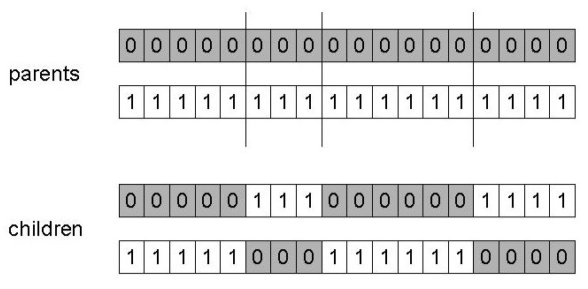
* Parents are selected by **proportional selection**: probability proportional to their fitness
* Implementation: roulette wheel
  + Assign each individual to a part of the roulette wheel, according to their fitness
  + Spin the wheel times to select individuals

### Crossover

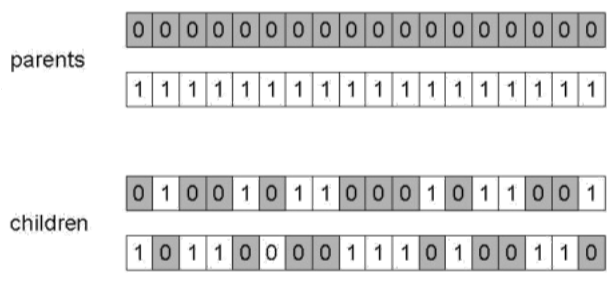
* Crossover is applied according to probability , typically in range (0.6, 0.9)
* **1-point crossover**:



* + Choose a random point
  + Split parents at crossover point
  + Create two children, by exchanging tails
* **N-point crossover**:



* + Choose random point
  + Split parents at crossover points
  + Create two children, by alternating parent traits
* **Uniform crossover**:



* + Treat each gene (bit) independently – decide which parent’s trait to take
  + Second child is inverse of first child

### Mutation

* Alter (bit-flip) each gene independently with probability (**mutation rate**), typically between and

Why crossover and mutation?

* Crossover is **explorative** – discover new areas in search space by crossing parent areas
* Mutation is **exploitative** – create random, small diversions, to optimize within promising area

## Genetic Algorithms: Alternative Approaches

### Alternative Representations

* **Gray coding** – (e.g. 0 = 000, 1 = 001, 2 = 011, etc.)
  + Hamming distance between any consecutive number is 1, means that small changes in genotype cause small changes in phenotype
* **Non-binary**: integers, real values, or permutations

### Population Models

* **Generational GA model (GGA)**:
  + Each individual survives for one generation
  + Entire set of parents is replaced by offspring
* **Steady-state GA model (SSGA)**:
  + Only part of population (usually one member) is replaced by offspring
  + **Generational gap**: proportion of population replaced
    - 1.0 for GGA (since all replaced)
    - for SSGA

### Parent and Offspring Selection

* **Parent selection**: select from current generation for mating
  + **Fitness-Proportionate Selection (FPS)**:
    - Probability of parent selection:
    - Expected number of copies of an individual :
      * = individual fitness
      * = average fitness
    - **Roulette wheel algorithm**: given probability distribution, spin a 1-armed wheel times to select individuals
      * No guarantee on value of
    - **Baker’s Stochastic Universal Sampling algorithm**: evenly-spaced arms on wheel, spin once
      * Guarantees
    - Problems:
      * Premature convergence – one highly-fit member can quickly take over
      * No selection pressure at end of runs where fitnesses are similar
      * Highly susceptible to function transposition
  + **Rank-Based Selection**:
    - Select probabilities on relative instead of absolute fitness
    - Rank population according to fitness; base selection probability on rank (fittest rank = , worst rank = )
  + Global population statistics, which above methods rely on, can be a bottleneck, and relies on external fitness function that may not exist
  + **Tournament Selection**:
    - Pick members at random, then select best one; repeat to select more individuals
* **Survivor selection**: select from parents + offspring for next generation
  + **Age-Based Selection**: used in GGA
  + **Fitness-Based Selection**: delete or replaced based on inverse of fitness
    - **Elitism**: always keep best individuals
    - **GENITOR**: delete worst individuals

## Real-Valued Genetic Algorithms

### Crossovers

* **Discrete**: use crossover operators from before, for binary representations
* **Intermediate**:
  + **Single arithmetic**: given
    - For parents and
      * Pick a gene at random
      * Child 1:
      * Child 1:
  + **Simple arithmetic**: given
    - Same as single arithmetic, but pick random gene and mix all genes after
    - For parents and
      * Child 1: replace each gene after with
      * Child 2: replace each gene after with
  + **Whole arithmetic**: given 
    - Replace all genes
    - For parents and
      * Child 1:
      * Child 2:

### Mutations

* **Uniform mutation**:
  + Choose a gene to change,
  + Choose random value between range of possible values
  + Replace
* **Add random noise**:
  + is random Gaussian number with mean 0 and standard deviation

## Permutations Genetic Algorithms

* Problems that involve deciding on the order of a sequence of events
  + **Order** of events is important – e.g. limited resources or time
  + **Adjacency** of events is important – e.g. TSP
* With variables, the representation is a list of integers, each of which occur exactly once

### Permutation GAs – Crossovers

* Four crossover operators:
  + **Adjacency-based**: PMX, edge crossover
  + **Order-based**: order 1 crossover, cycle crossover
* **Partially Mapped Crossover (PMX)**
  1. Copy over random segment from P1
  2. Starting from the first gene of P2 in the segment, find the value of the corresponding segment in P1, look for it in P2, and put it there
  3. Fill in the remaining elements from P2

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 3 | 7 | 8 | 2 | 6 | 5 | 1 | 4 |

Child 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 3 | 2 | 4 | 5 | 6 | 7 | 1 | 8 |

Child 2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 7 | 8 | 2 | 6 | 5 | 4 | 9 |

* **Order 1 Crossover**
  + Preserve relative order that elements occur
  1. Choose arbitrary part from P1 and copy to child
  2. Starting from cut point of copied part, copy elements in order of P2, wrapping around end (and skipping existing)

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 3 | 7 | 8 | 2 | 6 | 5 | 1 | 4 |

Child 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 2 | 1 | 4 | 5 | 6 | 7 | 9 | 3 |

Child 2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 7 | 9 | 8 | 2 | 6 | 5 | 1 | 3 |

* **Cycle Crossover**
  + Each allele comes from one parent together with its position
  + Make cycle of alleles from P1:
    - Start with first allele of P1, look for allele in same position in P2, go to position with same allele in P1
    - Repeat until get back to first allele of P1
  + Put alleles of cycle in positions of first parent
  + Take next cycle from second parent, and alternate

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 3 | 7 | 8 | 2 | 6 | 5 | 1 | 4 |

Child 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 7 | 4 | 2 | 6 | 5 | 8 | 9 |

Child 2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 2 | 3 | 8 | 5 | 6 | 7 | 1 | 4 |

### Permutation GAs – Mutations

* Four mutation operators:
  + **Insert mutation**
    - Pick two genes at random, shift genes so that second one follows the first

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 7 | 4 | 2 | 6 | 5 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 7 | 4 | 6 | 5 | 8 | 9 |

* + **Swap mutation**
    - Pick two genes at random, swap positions

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 7 | 4 | 2 | 6 | 5 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 7 | 4 | 3 | 6 | 5 | 8 | 9 |

* + **Inversion mutation**
    - Pick two genes, invert substring between them

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 7 | 4 | 2 | 6 | 5 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 7 | 3 | 6 | 5 | 8 | 9 |

* + **Scramble mutation**
    - Pick two genes, randomly rearrange genes between them

E.g.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 7 | 4 | 2 | 6 | 5 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 7 | 6 | 5 | 8 | 9 |

## When are GAs useful?

* Functions with many maxima
* Discrete/discontinuous functions
* High-dimensionality functions
* Non-linear parameter dependencies

## Difficulties with GAs

* Premature convergence
* Unable to overcome deception
* Need more evaluations than time permits
* Bad match of representation, mutation, and crossover makes operators destructive
* Biased/incomplete representation

## Adaptive Genetic Algorithms

* One of the main advantages of GA is **parameter tuning** – finding suitable values for algorithm parameters
  + Time-consuming
  + Parameters may not be independent – sub-optimal if tuned one-by-one
  + Simultaneous tuning can lead to too many experiments
  + Different values may be optimal at different stages
* Adaptive parameter control can be used to set parameters on the fly:
  + **Deterministic**: replace parameter with function , generation number
    - Doesn’t account for state of search
    - E.g. in Gaussian mutation,
      * Can set generation number, ranging from 0 to
      * starts at 1 and ends at 0.1 – helps enhance fine-tuning of search towards the end
  + **Adaptive**: use feedback from current state to heuristically control parameter values
    - **Rechenberg’s 1/5 success rule**:
    - Select based on rule – if ratio > 1/5, increase; if ratio < 1/5, decrease
  + **Self-adaptive**: incorporate parameter values into chromosomes, to drive parameter selection by selection, crossover, and mutation
    - Evolve as a chromosome
* Crossover operator can also be adapted:
  + **Top level**: adapt operator itself
  + **Medium level**: adapt probability of crossover
  + **Bottom level**: adapt crossing position or swapping probability
  + E.g. vector is calculated after each generation – frequency of 1-bit at each gene
  + of swapping different genes:
    - As GA progresses, 1-intrinsic and 0-intrinsic values will converge
    - As their frequencies approach 1 or 0, their swapping probability will be very low

## Parallel Genetic Algorithms

* **Master-slave approach**
  + Selection and mating handled by master processor, applied on entire population
  + Fitness evaluation distributed among slave processors
* **Fine-grained GAs**
  + Selection and mating restricted to local neighbourhoods that may overlap
  + Good for highly parallel processors
* **Coarse-grained GAs**
  + Multiple populations evolve in parallel
  + Selection and mating restricted to within populations
  + Populations exchange individuals every so often

### Coarse-Grained (Multiple-Deme) GAs

* Factors to consider:
  + **Topologies**: which populations are connected to each other – only connected populations can exchange individuals (migration)
    - Defines how fast solutions migrate
      * Dense topologies: good solutions spread faster, take over faster
      * Sparse topologies: solutions spread slower, so demes are more isolated
        + Allows different solutions to appear, which come together at the end to maybe produce better solution
  + **Migration policies**: how migrating individuals are selected and handled in receiving population
    - Best-Worst
    - Best-Random
    - Random-Random
    - Random-Worst
  + **Migration frequencies**: when migration occurs
    - Synchronous: migration every predetermined number of generations
      * Communication in early stages can lead to sub-optimal solutions and high communication costs
    - Asynchronous: migration after a certain event occurs (e.g. populations converged)
  + **Migration rates**: number of individuals per migration
    - Low migration rates cause demes to behave independently, so no significant effect of migration
    - High migration rates can cause premature convergence

### Cooperative GAs

* Multiple populations; fitness of individuals in each population depends on best individuals of other populations
* Different populations optimize different variables of the problem
* Works best if problem variables are independent

# Swarm Intelligence

* Interactions between agents result in collective problem-solving strategy
  + Direct or indirect contact
  + Stigmergy: indirect coordination between agents, through the environment
* **Emergent behaviour**: behaviour of system doesn’t depend on individuals, but their interactions
  + Flexibility: system performance adapts to internal/external changes
  + Robustness: system performs even if some individuals fail
  + Decentralization: distributed control
  + Self-organization: global behaviours from individual interactions
* Models of behaviour:
  + Swarm: coherent group, low level of polarization (parallel alignment)
  + Torus: entities rotate around empty core, random direction of rotation
  + Dynamic parallel group: coherent group, polarized entities that move around the group – density of group form fluctuates
  + Highly parallel group: coherent group, polarized entities with minimum fluctuations
* Five basic principles:
  + Proximity: carry out simple space and time computations
  + Quality: respond to quality environment factors
  + Diverse response: do not commit activities along narrow channels
  + Stability: do not change behaviour mode every time environment changes
  + Adaptability: change behaviour mode to benefit computational price

# Ant Colony Optimization

* Ants walking to or from a food source, deposit **pheromones**, chemical substance that influence other ants’ choice of a path
  + Pheromone trails help ants reach food sources identified by other ants – acts as **collective memory** for ants to communicate through
  + Pheromones **evaporate** over time, changing the environment

## ACO Algorithm

Given a graph with vertices and edges, where is the distance of edge between nodes and :

* Initialize each edge with **pheromone**
* Place ants at start node:
  + At each node , ant chooses to move to any of the nodes connected to it
  + Each node has probability of being selected by an ant :



* + and are chosen to balance **local** and **global** search ability
* Pheromone is **evaporated**:
  + is the pheromone decay factor
* Ant deposits extra pheromone on the path it chooses:
  + This increases the probability of subsequent ants choosing the same edge
* Termination conditions:
  + Max number of iterations reached
  + Acceptable solution reached
  + Stagnation – all ants are following the same path

### Choosing

* **Ant density model**: add a constant value – final pheromone is proportional to number of ants choosing it
* **Ant quantity model**: add – accounts for edge length, enforcing local search ability
* **Online delayed/Ant cycle model**: after ant builds the solution, it traces the path backward and updates the pheromone trails on the visited arc, according to solution quality



* + , where is the length of the path found by ant

### ACO Parameters

* **Number of ants**: more ants = more computation but more exploration
* **Max number of iterations**: enough to allow convergence
* **Initial pheromone**: constant, random, max value, or small value
* **Pheromone decay factor**:
* (global pheromone update factor) and (relative importance of visibility heuristic)

### ACO Components

* Transition rule – probability of selection by ant
* Pheromone evaporation rule
* Pheromone update rule
* Problem heuristic
* Measuring quality of solution
* Memory/list of constraints (Tabu list)
* Termination criteria

## Ant System (AS) Algorithm

* Same steps as basic ACO, with **online delayed** pheromone update:

## Ant Colony System (ACS) Algorithm

Based on AS, except:

* Transition rule based on **elitist strategy** – bias towards choices of better quality, controlled by

If – choose best quality solution  
Else,

* Local pheromone update – ants only update last traversed edge
* Pheromone update rule – uses best solution/ant only (iteration or global best)
* Candidate list
* Results show that not using pheromone deteriorates performance:
  + ACS without heuristics performs better than ACS without pheromone – likely due to ACS without heuristics is still guided by global update rule
  + ACS without pheromone reduces to a stochastic multi-greedy algorithm
  + ACS with both is better – confirms benefits of cooperation

## Max-Min Ant System (MMAS) Algorithm

* Pheromone update rule – done using best solution/ant in current iteration or overall, also decay in pheromone update
* Pheromone values are restricted between and – allows high exploration in the beginning, and more intensification later
* Improved performance over AS – overcomes stagnation

## ACO Characteristics

### Problem Characteristics

* Combinatorial optimization problems
  + Large problem size – infeasible to solve with classic optimization techniques
* Typically, discrete optimization problems

### Advantages of ACO

* Stochastic, population-based algorithm
* Retains memory of entire colony, instead of previous generation (GA)
* Random path selection means less affected by poor initial solutions
* Can handle dynamic environments

### Disadvantages of ACO

* Mainly experimental – many parameters, whose values are selected experimentally
* May take long time to converge

## ACO Adaptation

### ACSGA-TSP

* Run GA on top of ACS, to optimize parameter values
  + – greedy vs. probabilistic selection
  + – local pheromone update factor
  + – relative importance of visibility heuristic
* Each ant has own values of ACS parameters
* Chromosomes are randomly initialized
  + Single simple crossover
  + Bitwise mutation – mutate each bit based on given probability
* For each generation:
  + Choose best 4 individuals
  + For each best individual:
    - Run ACS-TSP with given values, record result as fitness of individual
  + Global pheromone update done by ant with best result
  + Choose 2 best individuals from the 4
  + Produce 2 children by applying crossover
  + Mutate 2 children
  + Replace worst 2 individuals with 2 children

### Near Parameter Free ACS

* Adapt parameters of ACS using same approach as adapting problem (ant approach)
  + – greedy vs. probabilistic selection
  + – local pheromone update factor
  + – global pheromone update factor
  + – relative importance of visibility heuristic
* Each ant selects suitable parameter values and next solution component
* Each ant’s parameter values are adaptively selected:
  + Initial value: middle of interval
  + Separate pheromone matrix for learning these values  
     specific parameter, range of values
    - For each parameter, interval is discretized with step , dividing interval into divisions
    - Parameter division is chosen based on pheromone
    - Parameter value:
      * lower bound of parameter value
      * upper bound of parameter value
  + No heuristic information is used

## ACO Cooperation

* **Heterogenous approach**: ants in two colonies have different behaviour (optimization criteria)
  + Applicable to multi-criteria optimization problems
* **Homogenous approach**: all ants have similar behaviour
  + Fine-grained: each processor holds one ant
  + Coarse-grained: each processor holds a colony
* Multi-colony algorithm, where all colonies are connected in directed ring:
  + All colonies get global best solution
  + Circular exchange of local best solutions – found to be best
  + Circular exchange of number of ants
  + Mixture of previous approaches

# Particle Swarm Intelligence

* Simulate collective behaviour of animals – individuals have no knowledge of global behaviour of group, they move together based on social interactions of neighbours
  + **Separation**: each agent tries to move away from nearby mates if they are too close (**collision avoidance**)
  + **Alignment**: each agent steers towards average heading of nearby mates (**velocity matching**)
  + **Cohesion**: each agent tries to go towards average position of nearby mates (**position control**)
* **Roost**: memory of previous own best and neighbourhood best positions
  + All individuals are attracted to the roost
  + Each memorizes position where it was closest to the roost
  + Each shares its information with others
  + By the end, all individuals land on the roost

# Particle Swarm Optimization

* **Particle**: individual/candidate solution
* **Swarm**: population

## PSO Motion

* Each particle holds essential movement information:
  + current position
  + current velocity
  + personal best position achieved so far
  + neighbourhood best position achieved so far
    - if neighbourhood is entire swarm
    - if neighbourhood restricted to few particles
* Each particle adjusts velocity to move towards personal best and neighbourhood best
* **Equations of motion:**



* + - velocity of particle (particle #, dimension) at iteration
    - position of particle at iteration
    - inertia weight
    - acceleration coefficients
    - random numbers
  + **Inertia** accounts for the fact that particles cannot suddenly change their direction of motion
  + are the weights in which each particle trusts its own experience (**cognitive component**) and its swarm experience (**social component**)
  + Random numbers are generated for each dimension, not each particle – if function optimizing for has 3 variables, the particle will have 3 dimensions
  + Important to set **maximum velocity** :
    - Too high, particles will go past optimal solutions
    - Too low, particles get stuck in local optima
* After, each particle **updates its personal best**:
* After, each swarm **updates global best**:

## PSO Algorithm

### Synchronous Update

* Initialize swarm
* While termination criteria not met:
  + For each particle:
    - Update particle velocity
    - Update particle position
    - Update personal best
  + Update neighbourhood best

### Asynchronous Update

* Initialize swarm
* While termination criteria not met:
  + For each particle:
    - Update particle velocity
    - Update particle position
    - Update personal best
    - Update neighbourhood best
* Tends to produce better results, since particles use more up-to-date information

### Termination Criteria

* Max number of iterations
* Max number of function evaluations
* Acceptable solution found
* No improvements over iterations

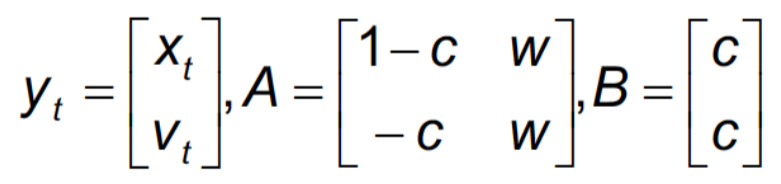
## PSO Neighbourhoods

* Selecting a proper neighbourhood helps convergence and avoiding getting stuck at local minima
  + **Star topology**: model
    - Each particle is influenced by all other particles
    - Fastest propagation of information in a population
    - Particles can get stuck in local minima easily
  + **Ring topology**: model
    - Each particle only influenced by particles in its neighbourhood
    - Slowest propagation of information
    - Increased computational cost, but doesn’t get stuck as easily in local minima
  + **Square topology**: most successful
* Most obvious way of choosing neighbours: pick particles physically closest in the search space
  + Can be computationally expensive, since distances must be computed each time particle changes position
* Can also store particles in matrix structure, and neighbours are particles stored next to a particle in the matrix (social neighbours)
* Neighbourhood size also affects performance

## PSO Initialization

* **Positions**: initialize randomly
* **Velocities**: initialize to 0 or small values
* initialize to initial position
* typically,
  + : social-only model
  + : cognition-only model – particles are individual hill climbers
  + Small values result in smooth trajectories
  + Large values result in more abrupt movement, towards or past good regions
* balances exploration and exploitation
  + Small values promote exploitation – gives more control to social, cognitive components
  + Large values promote exploration

## PSO Convergence

* For a one-dimensional system, let
* Let:
* Then:
  + The equations are in the form   
    
* By analyzing the equation, the conditions needed for stability are:

## Discrete PSO

* PSO was originally developed for continuous-valued spaces
* **Discretization** of position vectors or **redefining arithmetic operations** can make it work for discrete-valued spaces

### Binary PSO

* Each particle represents a position in the binary space – each element is 0 or 1
* Velocities are probabilities that an element will be in one state or the other
  + Hence, velocities
  + **Sigmoid function**:
* **Position update**:
  + is a randomly-generated number
* That means that the probability of
* Compared with three versions of GA, binary PSO was the only algorithm that found the global optimum during every single trial

### Permutation PSO

* Position of particle = solution to a problem (permutation of cities)
* Velocity of particle = sequence of swaps to be performed
* Three operations defined for new search space:
  + Adding velocity to position: applying sequence of swaps in velocity vector to position vector
  + Subtracting two positions: produces velocity – sequence of stops that transforms one position to the other
  + Multiplying velocity by a constant: changes length of velocity vector according to constant
    - length set to zero
    - velocity is truncated
    - velocity is augmented
* Using a **social neighbourhood** is less expensive than **physical neighbourhood**
* Can be computationally expensive to apply to larger problems

### Great Value Priority

* Different PSO approach – perform **space transformation** from a particle in continuous domain to a solution space permutation
* Steps:
  + Sort all elements in position vector in descending order
  + Take sorted indices as permutation
  + If had the highest value in the position vector, then comes first in the permutation vector
* E.g.

– used to calculate fitness

* This transformation occurs before calculating fitness
* Can also use local search on permutation, then transform back to vector
  + E.g. after local search  
     (swap same elements)

## Particle Swarm Optimization – Adaptation

### TRIBES

* **Parameter-free** PSO that adapts number of particles used in search
* **Tribe**: group of connected particles
  + All tribes are connected in some way, to inform one another of their findings



* + This helps decide global minimum among solutions found by different tribes



* A particle is **good** if its improved in the last iteration, otherwise it is **neutral**
* Each particle memorizes last two solutions; a particle is **excellent** if last two were both improvements
* number of good particles in tribe
* total number of particles in tribe
  + Good tribes delete their worst particle
  + Every bad tribe randomly generate a new particle; new particles form new tribe
    - Each particle is connected to the tribe that generated it, through its best particle
* Idea: start with single particle
  + If no improvement, larger and larger tribes generated
  + If improvements, good tribes remove their worst particles
* No clear relation between number of particles and performance

## Particle Swarm Optimization – Cooperation

### Concurrent PSO

* Update two different swarms in parallel, each using different algorithms
* Swarms exchange every predetermined number of iterations
* Both swarms track the better

### Cooperative PSO

* Multiple swarms, fitness of any particle in a swarm depends on the particles in other swarms
* Different swarms optimize different problem variables/dimensions
* Fitness of a particle is determined by its value and value of the best particles in other swarms
* Performs best if problem variables are independent

### Hybrid Cooperative PSO

* Two swarms – one uses PSO, the other swarm uses cooperative PSO (CPSO), serial updates
  + When PSO swarm is updated, sends to CPSO swarm
  + CPSO swarm uses received to update random particles of its sub-swarms
  + When CPSO swarm is updated, it sends context vector to PSO swarm
  + PSO swarm uses context vector to replace a random particle

## Particle Swarm Optimization – Applications

### FPGA Placement

* Minimize total wire length
* Each particle contains information of all available locations
* Velocities = swap pairs
* PSO converges faster than SA, but has higher computational cost

### Neural Network Training

* **Neural network**: performs mapping from set of input data to one or more output nodes
  + Partially or fully connected set of nodes that are stimulated by inputs and produce outputs that may simulate other nodes
  + Find appropriate set of weights to lead to satisfactory network behaviour
* PSO proposed for optimizing weights of an ANN performing the XOR operation
  + Each particle and velocity are 9-dimensional vectors
  + Objective: minimize error between desired and actual outputs of network

### Clustering

* **Classification**: function that assigns an object to a class
  + Supervised
  + Uses labelled data
  + Requires training phase
  + Domain-sensitive
* **Clustering**: given objects, group them into groups such that all objects in a single group have a “natural” relation to one another
  + Unsupervised
  + Uses unlabelled data
  + Organize patterns according to optimization criteria
  + Notion of similarity
* **Set partitioning**: a partition is a grouping of the set’s elements into non-empty subsets, such that each element only appears in one subset
* **Goal**: minimize intra-cluster distances, maximize inter-cluster distances