Subhranil Sarkar

Artificial Intelligence and Data Mining

2nd Semester | M.Sc. Data Science

Roll: 96/DTS No.: 210018

Reg. No.: 020873 of 2018-2019

University of Kalyani

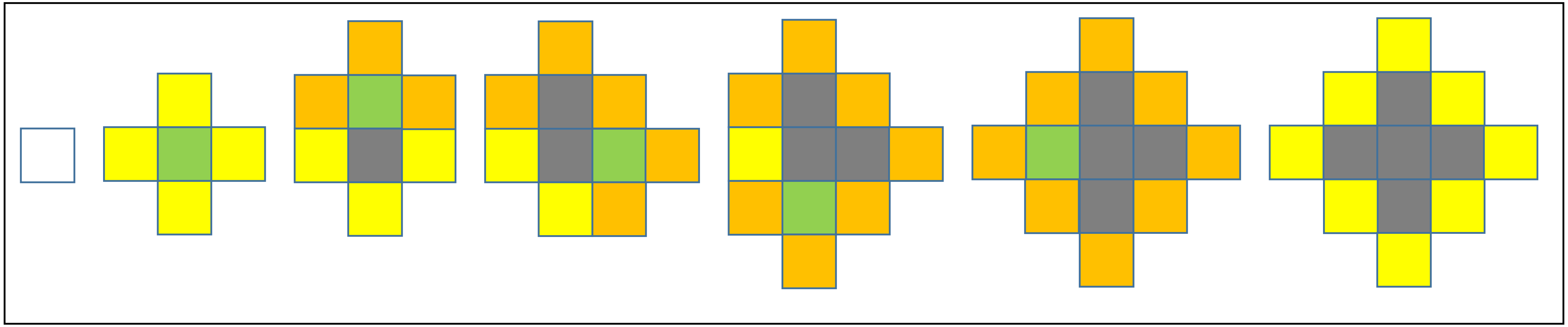
Index

- Breadth First Search
- Depth First Search
- Genetic Algorithm
- Decision Tree
- Support Vector Machine

Breadth First Search

Process

- Breadth first Search is a simple graph-search technique.
- Root node is expanded first, then all the successors of root node are expanded next, then their successors, and so on.
- All the nodes at a certain depth are expanded first before any nodes at the next level. That's why it is called "Breadth" First Search.



Process

- Shallowest unexpanded node is always chosen for expansion.
- This is done by using a FIFO queue at the frontier.
- Thus new nodes go to the back of the queue, and old ones, which are shallower than the new nodes, get expanded first.
- Always checks if the generated nodes are goal node or not.
- Using explored set to store all the visited nodes.
- Using frontier and explored set we always get the shallowest path in this search algorithm.

Implementation

```
• function BFS(graph, start, goal) returns a solution, or failure
• node ← start state, path-cost = 0
• if initial is goal then return node
• frontier ← FIFO queue with the node
• explored ← an empty set
• loop do
•     if isEmpty(frontier) then return failure
•     node ← pop(frontier)  /* shallowest node */
•     explored.add(node)
•     for each child in expand(node) do
•         if child is not in explored or frontier then
•             if child is goal then return path(start, child)
•             frontier.insert(child)
```

Depth First Search

Process

- Depth-First Search is a graph-search algorithm.
- Always expands the deepest node in the current frontier.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- After expanding the nodes, they are dropped from the frontier.
- So the previous deepest node that still has unexplored successors will expand.

Process

- The most recent unexpanded node is always chosen for expansion.
- This is done by using a LIFO queue at the frontier.
- Two versions of DFS are there: graph-search and tree-search depending on whether the explored states are recorded or not.
- Graph-search version is complete in finite space.
- DFS is nonoptimal.

Implementation

```
• function DFS(graph, start, goal) returns a solution, or failure
• node ← start state, path-cost = 0
• frontier ← LIFO queue with the node
• explored ← an empty set
• loop do
•     if isEmpty(frontier) then return failure
•     node ← pop(frontier)  /* shallowest node */
•     if node is goal then return path(start, node)
•     explored.add(node)
•     for each child in expand(node) do
•         if child is not in explored or frontier then
•             frontier.extend(child)
```

Genetic Algorithm

Process

- Genetic Algorithm is inspired by Charles Darwin's theory of natural evolution.
- Successor states are generated by combining two parents.
- **Chromosome:** randomly generated binary string of a random size.
- **Population:** *k* randomly generated chromosomes.
- **Fitness:** From a given function we have to evaluate fitness of a chromosome.
- **Selection:** Select best chromosomes and create a mating pool.

Process

- **Crossover:** Select two random chromosomes and select a random point in the chromosome and exchanged parts of the chromosomes to create children.
- **Mutation:** With a mutation probability, flip a bit from the chromosome, which resembles mutation in the biology.
- **Elitism:** Carry the parent to the new population if the greatest fitness of the newly generated population is less than the previous population.

Implementation

- function GA(popSize, chromLen, maxIter, crossProb, mutProb)
- population ← makePopulation(popSize, chromLen)
- fit ← fitness(population)
- for (i = 0; i < maxIter; i = i + 1) do
-     matingPool ← selection(population)
-     crossPopulation ← crossover(matingPool, crossProb)
-     mutatedPopulation ← mutation(crossPopulation, mutProb)
-     childBestFit ← fitness(mutatedPopulation)
-     oldBest ← best(fit)
-     newBest ← best(childBestFit)
-     if oldBest >= newBest then exchange(oldBest, newBest)
-     population ← mutatedPopulation
-     fit ← childBestFit
- return bestChrom(population)

Decision Tree

Process

- Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems.
- A **decision tree** is a flowchart-like tree structure, where each **internal node** (non-leaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label.
- The topmost node in a tree is the **root** node.
- The decisions or the test are performed on the basis of features of the given dataset.
- In order to build a tree, we use some algorithms which are **ID3**, **C4.5**, **CART**. This is called Decision Tree Induction.
- ID3, c4.5, CART adopt a greedy approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner.

Process

- To determine the best split, using greedy approach nodes with homogeneous class distribution are preferred.
- Measures of Node impurity are:
  - Gini Index
  - Entropy
  - Gain Ratio
- Stopping Criteria for tree induction:
  - Stop expanding a node when all the records belong to the same class.
  - When all the records have similar attribute value.

Implementation

```
• create a node N;
• if tuples in D are all of the same class, C, then
•    return N as a leaf node labeled with the class C;
• if attribute list is empty then
•    return N as a leaf node labeled with the majority class in D; // majority voting
• apply Attribute selection method(D, attribute list) to find the “best” splitting criterion;
• label node N with splitting criterion;
• if splitting attribute is discrete-valued and
•    multiway splits allowed then // not restricted to binary trees
•    attribute list ← attribute list − splitting attribute; // remove splitting attribute
• for each outcome j of splitting criterion
•    // partition the tuples and grow subtrees for each partition
•    let Dj be the set of data tuples in D satisfying outcome j; // a partition
•    if Dj is empty then
•       attach a leaf labeled with the majority class in D to node N;
•    else attach the node returned by Generate decision tree(Dj, attribute list) to node N;
• endfor
• return N;
```

Support Vector Machine (SVM)

Process

- Support Vector Machine (SVM) is a method for the classification of both linear and nonlinear data.
- Uses a nonlinear mapping to transform the original training data into a higher dimension.
- Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another).
- The SVM finds this hyperplane using *support vectors* and *margins* (defined by the support vectors).

Process

- SVM chooses the extreme points/vectors that help in creating the hyperplane.
- These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine.
- SVM can be of two types:
  - Linear SVM.
  - Non-linear SVM.

Implementation

Require: X and y loaded with training labeled data, $\alpha \Leftarrow 0$ or $\alpha \Leftarrow partially\ trained$

SVM

- C ⇐ some value (let 10)
- repeat
-     for all $\{x_i, y_i\}$, $\{x_j, y_j\}$ do
-         Optimize α_i and α_j
-     end for
- until no changes in α or other resource constraint criteria met

Ensure: Retain only the support vectors ($\alpha_i > 0$)