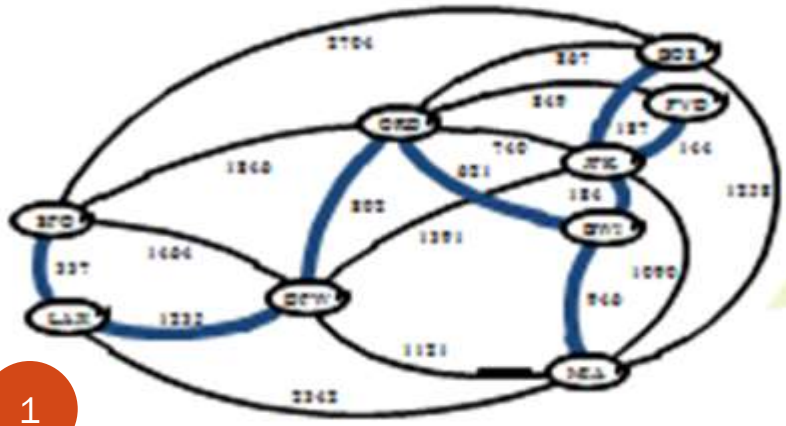
Course Title: Algorithm Analysis and Design



ALGORITHM ANALYSIS AND DESIGN

SRM UNIVERSITY AP Amaravati

- Unit I :Algorithmic thinking & motivation with examples, Reinforcing the concepts of Data Structures with examples.Complexity analysis of algorithms: big O, omega, and theta notation, Analysis of Sorting and Searching, Hash table, Recursive and non-recursive algorithms.
- Unit II:***General Problem Solving (GPS) techniques:***
- **Divide and conquer**: Merge sort, Quicksort, BST, Master method for Complexity analysis
- **Greedy method**: Fractional Knapsack, Minimum spanning trees (Prim's & Kruskal's), Shortest paths: Dijkstra's algorithm, Huffman coding
- **Dynamic Programming**: 0/1 Knapsack, All-to-all shortest paths

ALGORITHM ANALYSIS AND DESIGN

- Unit III: **BFS & DFS**, **Backtracking**: 8-Queens problem, Knights tour, Travelling Salesman Problem (TSP), **Branch-and-bound**: 16-puzzle problem, TSSP, **Randomized algorithms**: Playing Cards, Scheduling algorithms.
- UNIT IV
- **Pattern matching algorithms**: Brute-force, Boyer Moore, KMP algorithms.
- **Algorithm analysis**: Probabilistic Analysis, Amortized analysis, Competitive analysis.

UNIT V

- **Non-polynomial complexity**: examples and analysis, Vertex cover, Set cover, TSP, 3-SAT
- **Approximation Algorithms**: Vertex cover, TSP, Set cover

SRM UNIVERSITY AP Amaravati

Text books:

1. Cormen, Leiserson, Rivest, Stein, "Introduction to Algorithms", 3rd Edition, MIT press, 2009

2. Parag Dave & Himanshu Dave, "Design and Analysis of Algorithms", Pearson Education, 2008

Reference books:

- 1.Michel Goodrich, Roberto Tamassia, "Algorithm design-foundation, analysis & internet examples", Wiley., 2006
- 2.A V Aho, J E Hopcroft, J D Ullman, "Design and Analysis of Algorithms", Addison-Wesley Publishing.
- 3. Algorithm Design, by J. Kleinberg and E. Tardos, Addison-Wesley, 2005
- 4. Algorithms, by S. Dasgupta, C. Papadimitriou, and U. Vazirani, McGraw-Hill, 2006

ALGORITHM ANALYSIS AND DESIGN

	Assessment tool	Conducting Marks	Converting Marks	Final Conversion
Theory	Mid-term-I	25	15	30
	Mid-term-II	25	15	
	Assignments	10	10	
	Quiz	10	10	
Practical	Lab performance	20	20	20
	Observation note	10	10	
			Total	50

	Assessment tool	Conducting Marks	Final Conversion
End semester theory exam	Final exam	100	30
Assignments	Project	100	20
		Total	50

Introduction to C++

- Programming Concepts
- Basic C++
- C++ Extension from C

Focus

- Focus on
  - Algorithm Concepts
  - Algorithm Design Techniques

- Not to get lost in Language Details

What is programming?

Programming is taking

A ***problem***

Find the area of a rectangle

A set of ***data***

length

width

A set of ***functions***

*area = length * width*

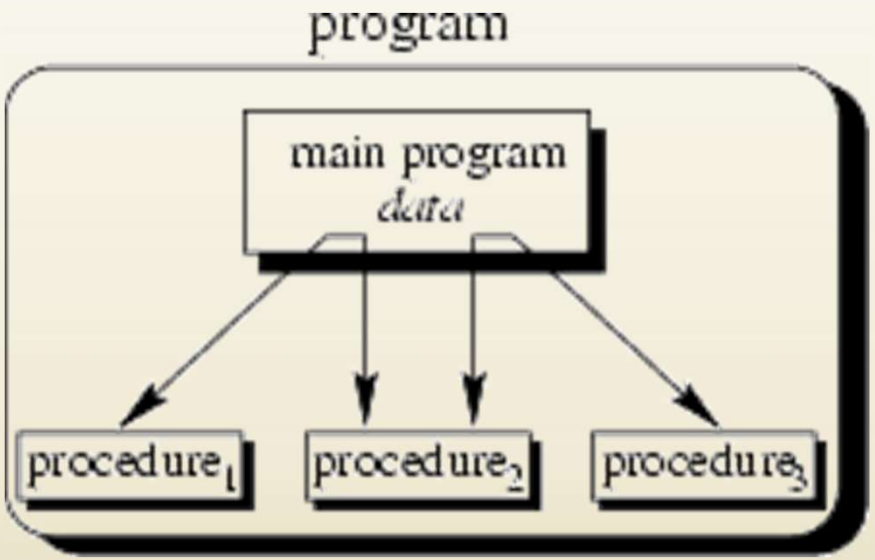
Then,

Applying functions to data to solve the problem

Programming Concept Evolution

- **Unstructured**
- **Procedural**
- **Object-Oriented**

Procedural Concept



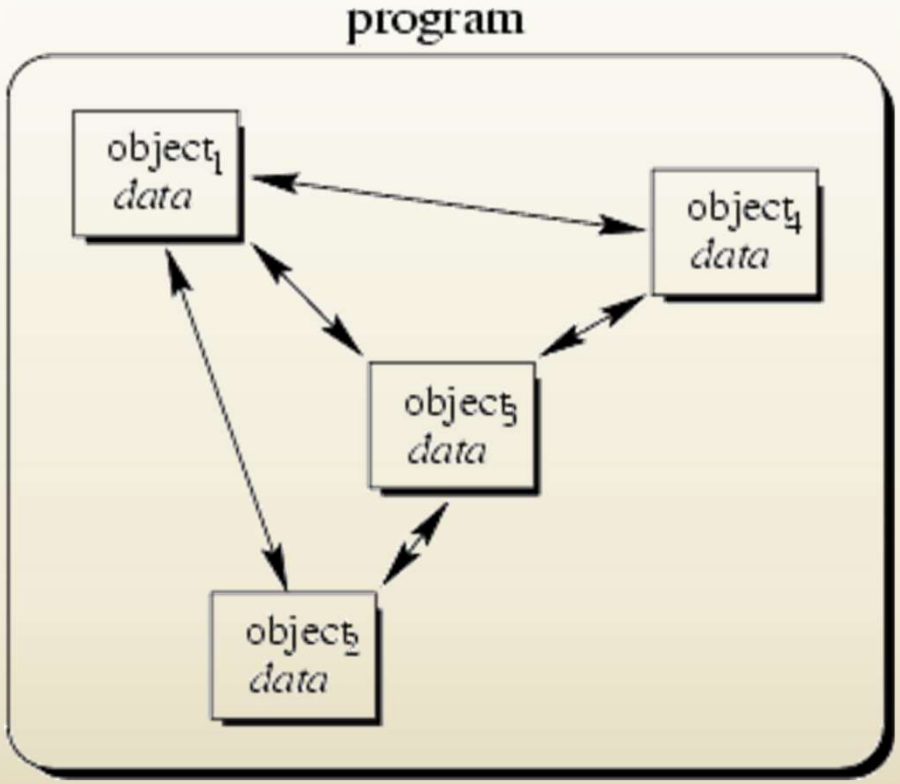
- The main program coordinates calls to procedures and hands over appropriate data as parameters.

Procedural Concept (II)

- Procedural Languages
  - C, Pascal, Basic, Fortran
  - Facilities to
    - Pass arguments to functions
    - Return values from functions
- For the rectangle problem, we develop a function

```
int compute_area (int L, int W)
{
        return ( L * W );
 }
```

Object-Oriented Concept



- Objects of the program interact by sending messages to each other

Objects

An object is an encapsulation of both functions and data

- ***Objects are an Abstraction***
  - *represent real world entities*
  - *Classes are data types that define shared common properties or attributes*
  - *Objects are instances of a class*
- ***Objects have State***
  - *have a value at a particular time*
- ***Objects have Operations***
  - *associated set of operations called methods that describe how to carry out operations*
- ***Objects have Messages***
  - *request an object to carry out one of its operations by sending it a message*
  - *messages are the means by which we exchange data between objects*

OO Perspective

Let's look at the Rectangle through object oriented eyes:

- Define a new type Rectangle (a class)
  - Data
    - width, length
  - Function
    - area()
- Create an instance of the class (an object)
- Request the object for its area

In C++, rather than writing a procedure, we define a class that encapsulates the knowledge necessary to answer the question - here, what is the area of the rectangle.

Example Object Oriented Code

```
class Rectangle
{
  private:
    int width, length;
  public:
    Rectangle(int w, int l)
        {
            width = w;
            length = l;
        }
    int area()
    {
        return width*length;
    }
};
```

```
main()
{
    Rectangle rect(3, 5);
    cout << rect.area()<<endl;
}
```

Object-Oriented Programming Languages

- Characteristics of OOPL:
  - Encapsulation
  - Inheritance
  - Polymorphism
- OOPLs support :
  - Modular Programming
  - Ease of Development
  - Maintainability

Characteristics of OOPL

- **Encapsulation:** Combining data structure with actions
  - Data structure: represents the properties, the state, or characteristics of objects
  - Actions: permissible behaviors that are controlled through the member functions

  Data hiding: Process of making certain data inaccessible
- **Inheritance:** Ability to derive new objects from old ones
  - permits objects of a more specific class to inherit the properties (data) and behaviors (functions) of a more general/base class
  - ability to define a hierarchical relationship between objects
- **Polymorphism:** Ability for different objects to interpret functions differently

Basic C++

- Inherit all ANSI C directives
- Inherit all C functions
- You don't have to write OOP programming in C++

Basic C++ Extension from C

- comments

```
/* You can still use the old comment style, */
/* but you must be // very careful about mixing them */
// It's best to use this style for 1 line or partial lines
/* And use this style when your comment
 consists of multiple lines */
```

- cin and cout (and #include <iostream.h>)

```
cout << "hey";
char name[10];
cin >> name;
cout << "Hey " << name << ", nice name." << endl;
cout << endl;      // print a blank line
```

- declaring variables almost anywhere

```
// declare a variable when you need it
   for (int k = 1; k < 5; k++){
       cout << k;
    }
```

Basic C++ Extension from C (II)+

- const
  - In C, #define statement
    - Preprocessor - No type checking.
    - #define n 5
  - In C++, the const specifier
    - Compiler - Type checking is applied
    - const int n = 5; // declare and initialize
- New data type
  - Reference data type "&".

```
int ix; /* ix is "real" variable */
int & rx = ix; /* rx is "alias" for ix. Must initialize*/
ix = 1; /* also rx == 1 */
rx = 2; /* also ix == 2 */
```

C++ - Advance Extension

- C++ allows function overloading
  - In C++, functions can use the same names, within the same scope, if each can be distinguished by its name *and* signature
  - The signature specifies the number, type, and order of the parameters expressed as a comma separated list of argument types

C++

- Is a better C
- Expressive
- Supports Data Abstraction
- Supports OOP
- Supports Generic Programming
  - Containers
    - Stack of char, int, double etc
  - Generic Algorithms
    - sort(), copy(), search() any container Stack/Vector/List

Reminder

- There are many different kinds of programming paradigms, OOP is one among them.
- In OOP, programmers see the execution of the program as a collection of dialoging objects.
- The main characteristics of OOPL include encapsulation, inheritance, and polymorphism.

MOTIVATION EXAMPLES

- Implement 8 queens problem. Analyse how the number of possibilities decrease with each constraint.
- Implement generate and test method for 16-puzzle problem. Propose a heuristic function and use it in the program.
- Create a pack of playing cards (4 suits (colors) & each suit will have 2,3,..10, Jack , Queen, King, Ace). Display the cards. Simulate shuffling of the pack and display the shuffled cards. Distribute them among 4 players in a Round robin fashion and display the cards with each player. Give some value for each color and card and find the total values of the cards with each player.

MOTIVATION EXAMPLES

- **Simulate the following scenario assuming that the time between calls is random. There are two technical support people: Able and Baker. Able is more experienced and can provide service faster than Baker. Times are usually a continuous measure, but this time-based example is made discrete for ease of explanation. The simulation proceeds in a complex manner because of two servers. When both are free Able gets the call. Assume service distribution other than given in your slides.**

Basic skills

* Memory model?
* Be familiar with linked lists: File as a linked list, Hash table as a set of linked lists
- Be familiar with stacks and recursion: Towers of Hanoi: Recursive & Non-recursive solution
- Be familiar with Queues: Car wash station, Scheduling algorithms in OS

PROBLEM SOLVING TECHNIQUES:

- *Generate & Test*
- *Hill Climbing*
- *Divide and Conquer*
- *Greedy method*
- *Dynamic Programming*
- *Backtracking*
- *Branch-and-Bound*

Generate-and-test

- Very simple strategy - just keep guessing.

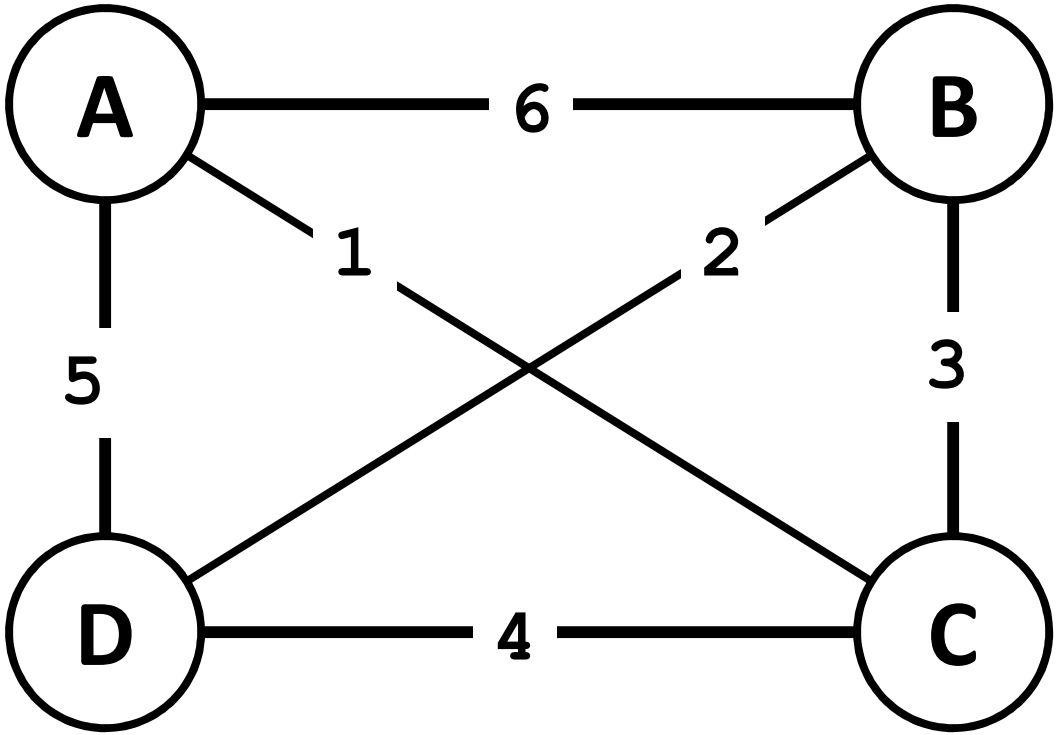
```
do while goal not accomplished
   generate a possible solution
   test solution to see if it is a goal
```

- Heuristics may be used to determine the specific rules for solution generation.

Example - Traveling Salesman Problem (TSP)

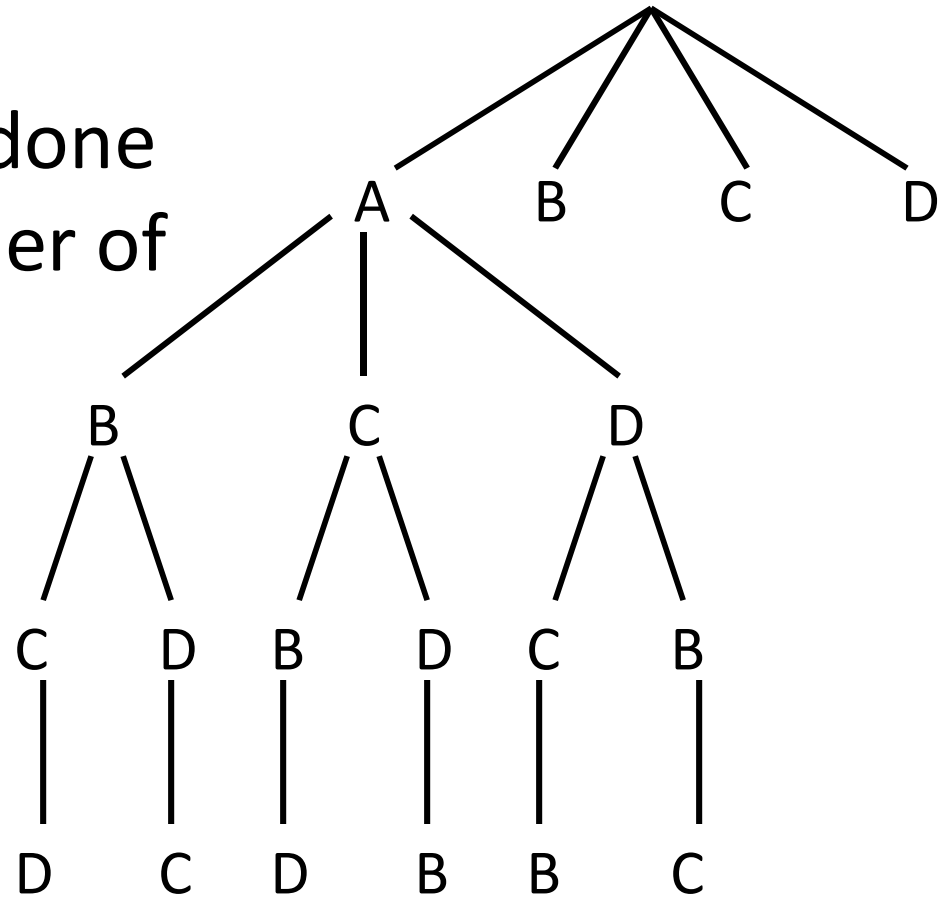
- Traveler needs to visit *n* cities.
- Know the distance between each pair of cities.
- Want to know the shortest route that visits all the cities once.
- *n=80* will take millions of years to solve exhaustively!

TSP Example



Generate-and-test Example

- TSP - generation of possible solutions is done in lexicographical order of cities:
  1. A - B - C - D
  2. A - B - D - C
  3. A - C - B - D
  4. A - C - D - B
  ...



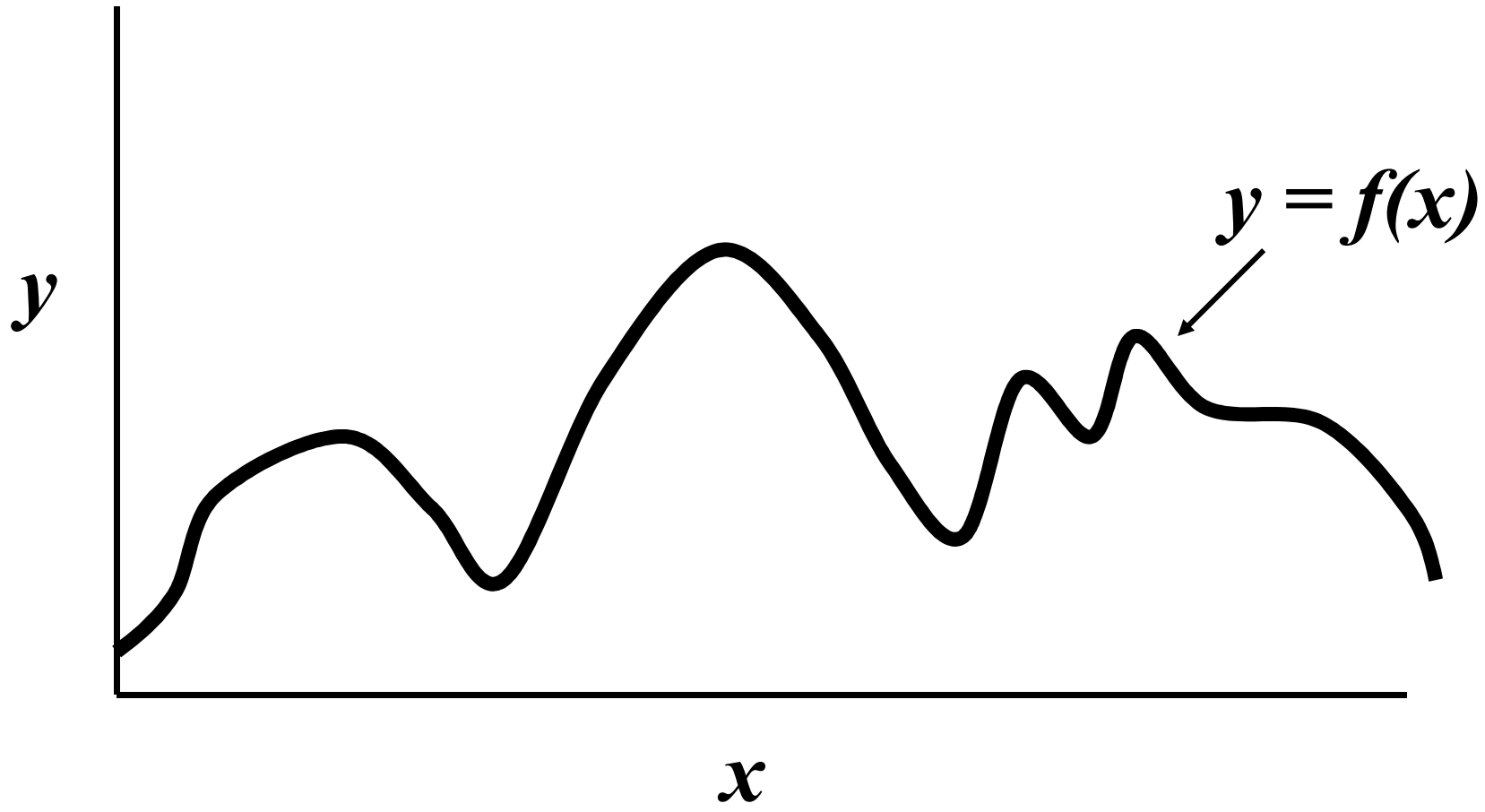
Hill Climbing

- Variation on generate-and-test:
  - *generation* of next state depends on feedback from the *test* procedure.
  - *Test* now includes a heuristic function that provides a guess as to how good each possible state is.
- There are a number of ways to use the information returned by the *test* procedure.

Simple Hill Climbing

- Use heuristic to move only to states that are *better* than the current state.
- Always move to better state when possible.
- The process ends when all operators have been applied and none of the resulting states are better than the current state.

Simple Hill Climbing Function Optimization



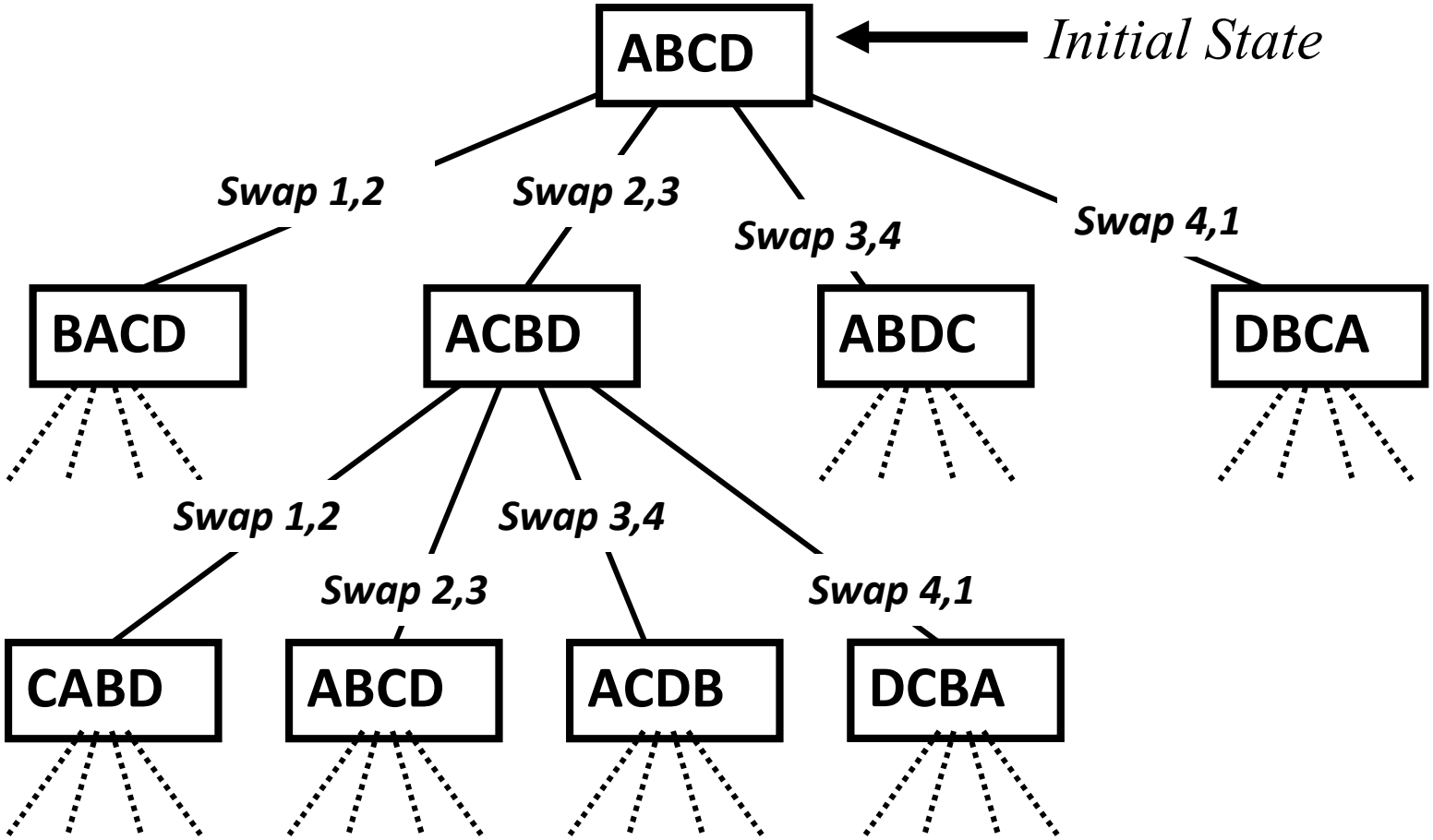
Potential Problems with Simple Hill Climbing

- Will terminate when at local optimum.
- The order of application of operators can make a big difference.
- Can’t see past a single move in the state space.

Simple Hill Climbing Example

- TSP - define state space as the set of all possible tours.
- Operators exchange the position of adjacent cities within the current tour.
- Heuristic function is the length of a tour.

TSP Hill Climb State Space



Steepest-Ascent Hill Climbing

- A variation on simple hill climbing.
- Instead of moving to the *first* state that is *better,* move to the best possible state that is one move away.
- The order of operators does not matter.

- Not just climbing to a better state, climbing up the *steepest* slope.

Stochastic hill climbing

- It does not examine all the neighboring nodes before deciding which node to select .
- It just selects a neighboring node at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

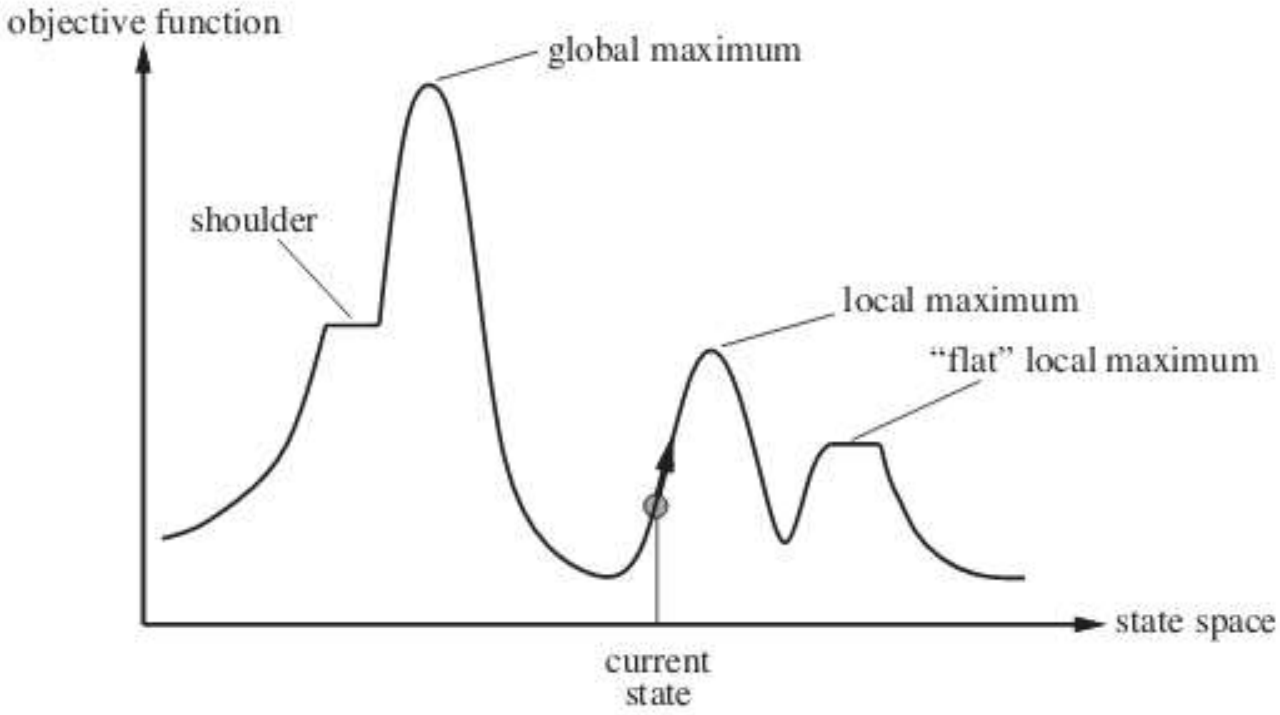
Hill Climbing Termination

- Local Optimum: all neighboring states are worse or the same.
- Plateau - all neighboring states are the same as the current state.
- Ridge - local optimum that is caused by inability to apply 2 operators at once.

Heuristic Dependence

- Hill climbing is based on the value assigned to states by the heuristic function.
- The heuristic used by a hill climbing algorithm does not need to be a static function of a single state.
- The heuristic can look ahead many states, or can use other means to arrive at a value for a state.

Heuristic Dependence



Best-First Search

- Combines the advantages of Breadth-First and Depth-First searchs.
  - DFS: follows a single path, don't need to generate all competing paths.
  - BFS: doesn't get caught in loops or dead-end-paths.
- Best First Search: explore the most promising path seen so far.

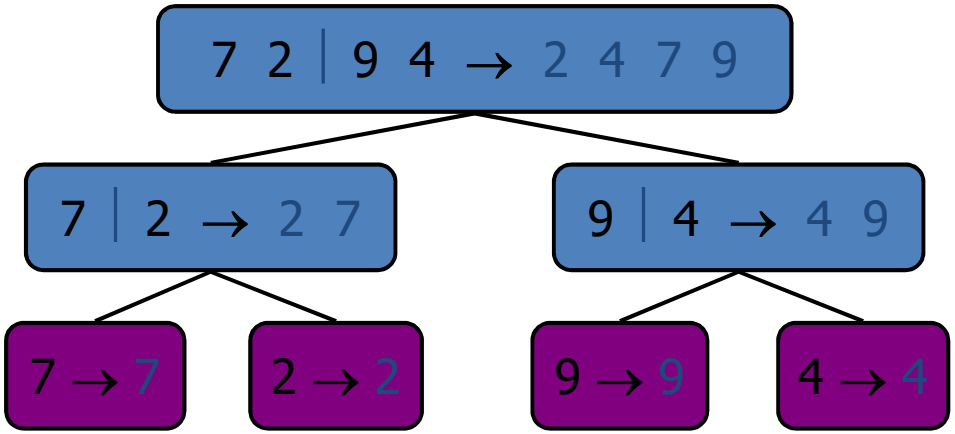
Best-First Search (cont.)

While goal not reached:

1. Generate all potential successor states and add to a list of states.
2. Pick the best state in the list and go to it.

- Similar to steepest-ascent, but don’t throw away states that are not chosen.

DIVIDE-AND-CONQUER



Merge sort, Quick sort
Binary Search

Greedy algorithm

- **A *greedy algorithm always makes the choice that looks best at* the moment. That is, it makes an optimal choice with locally available information in the hope that this choice will lead to a globally optimal solution.**

Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Greedy algorithms generate one solution sequence, thereby saving space and time.

Dynamic programming

Dynamic programming solves problems by combining the solutions to sub-problems. dynamic programming applies when the sub-problems overlap—that is, when sub-problems share sub-sub-problems. A dynamic-programming algorithm solves each sub-sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem.

We typically apply dynamic programming to *optimization problems. Such problems* can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Dynamic programming compute the value of an optimal solution, typically in a bottom-up fashion by keeping all the partial solutions in memory.

Hence this process always give the optimal solution but require more memory and computation.

Backtracking

- DFS + Constraints
- Eight queens problem
- Knight's Tour
- Rat in a Maze

Branch and Bound technique

- **BFS + Constraints**
- **16-puzzle**
- **Traveling salesman problem**

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	