COMBINATORIAL OPTIMIZATION PROBLEMS

Fundamentals of Artificial Intelligence

Session 10

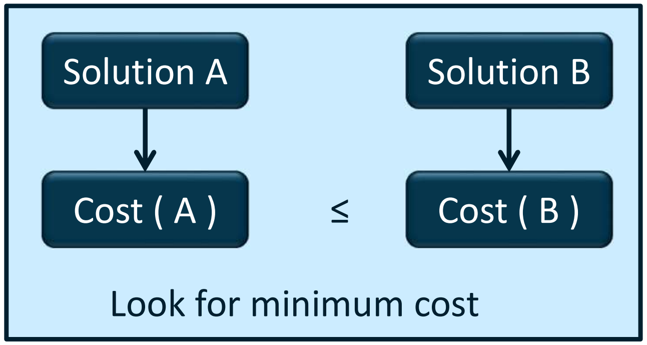
Pramod Sharma
pramod.sharma@prasami.com

Agenda

- Matrix Multiplication
- Recursion
- Dynamic Programming
- Branch and Bound

Combinatorial Optimization

- ❑ What do we optimize
  - ❖ A cost function-define cost of choosing an option
  - ❖ Optimization of cost function lets us define which option is better
  - ❖ Cost function is also defining ordering among solutions
  - ❖ Ordering can be defined in two ways
    - ➢ Total ordering
    - ➢ Partial ordering (falls under multi criteria optimization)

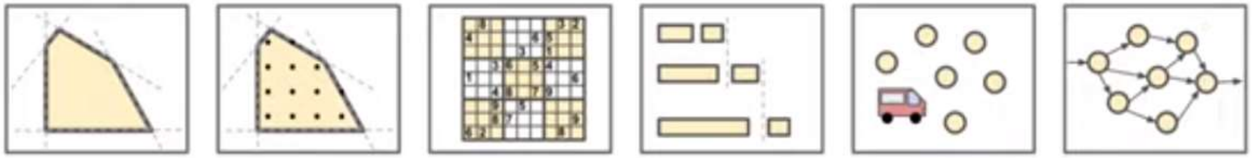

Solution A

Solution B

Cost (A) ≤ Cost (B)

Look for minimum cost

- ❑ What is included
  - ❖ Linear programming
  - ❖ Mixed Integer programming
  - ❖ Constrain programming
  - ❖ Airline scheduling
  - ❖ Cargo routing
  - ❖ Specialized graph algorithms



How to Find the Least Cost Solution

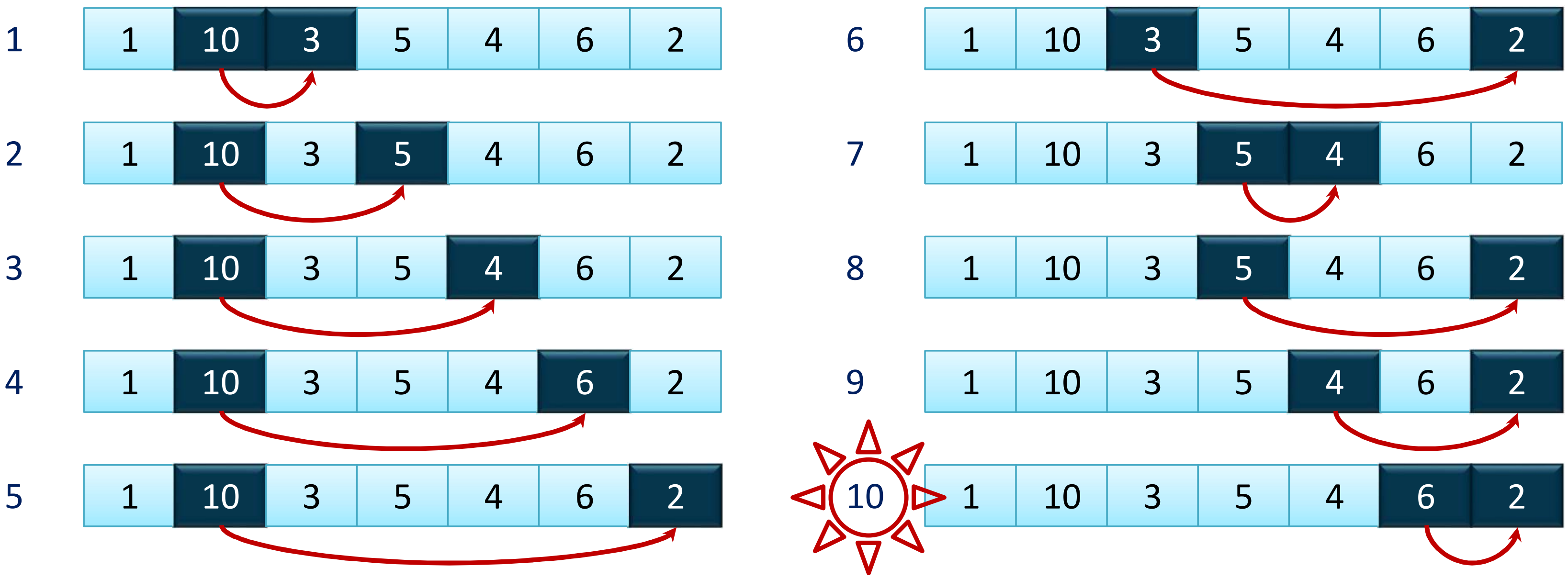
- Design an optimization algorithm
- Decompose the problem into smaller problems
  - We can deal with problems with optimal sub-structure
  - If substructure is not optima, we cannot find optimal solution
- How to search among alternatives?
  - Complexity
  - Execution time
- Combinatorial Explosion :
  - Number of alternatives to be explored grows very fast with the increase in the problem size

Example: Out-of-Order Pairs

- A pair is out-of-order if it is not in ascending order

1	10	3	5	4	6	2

- Let’s find out how many pairs are out of order



Example: Find out of order pairs

- ❑ A pair is out-of order if it is not in ascending order

1	10	3	5	4	6	2

- ❑ Reorder the elements to minimize the cost function
  - ❖ Look for out of order pair x_i, x_j
  - ❖ Swap (x_i, x_j)
  - ❖ Loop until is all sorted…

- ❑ Will it always converge?
  - ❖ Yes, as with each of the iteration we are reducing number of out-of-order pairs
  - ❖ It may take longer but eventually we will get it in order

- ❑ Brute force solutions are there, can we be little smart about it and do it with lesser effort?

Example: Find out of order pairs

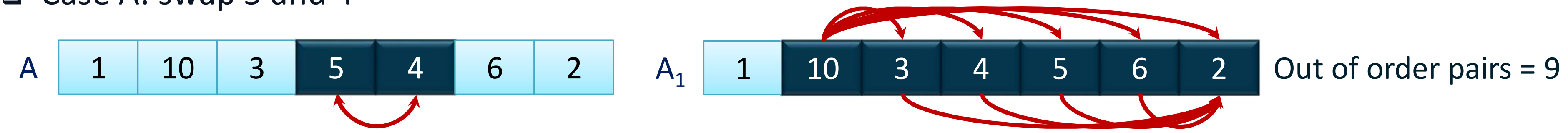
- A pair is out-of order if it is not in ascending order

1	10	3	5	4	6	2

Out of order pairs = 10

- A max 2 swaps allowed

- Case A: swap 5 and 4



A

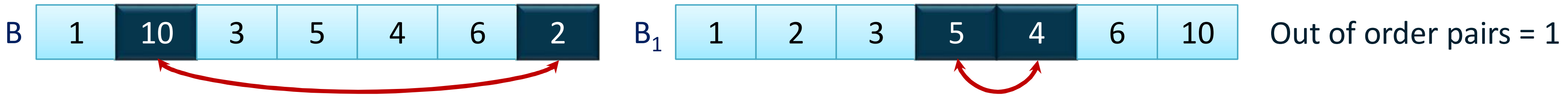
1	10	3	5	4	6	2

A_1

1	10	3	4	5	6	2

Out of order pairs = 9

- Case B : swap 10 and 2



B

1	10	3	5	4	6	2

B_1

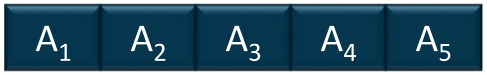
1	2	3	5	4	6	10

Out of order pairs = 1

Smart choices will help us reach the goal faster saving lot of compute time.

Example: Matrix Multiplication

- ❑ Given five matrices



A_1 A_2 A_3 A_4 A_5

  - ❖ Size of matrix A_i ≡ $x_i\,[A_i]\,x_{i+1}$ or Size of matrix A_i ≡ x_i * x_{i+1}

- ❑ If we multiply A_i and A_{i+1}
  - ❖ Cost of multiplication is x_i * x_{i+1} *x_{i+2}

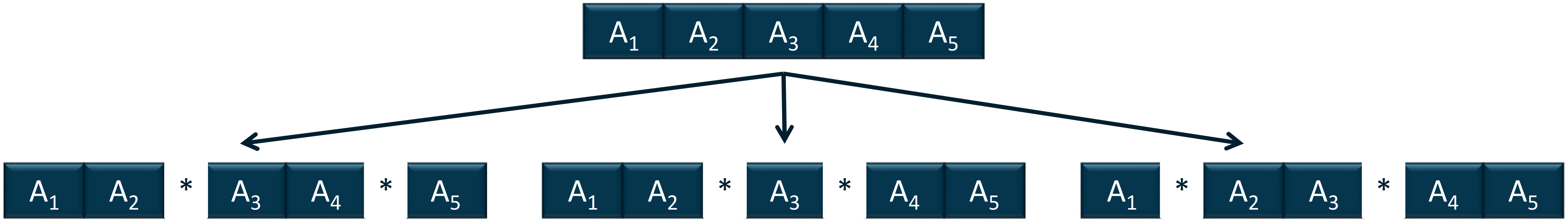
- ❑ How to multiply matrix with minimum cost?

Example: Matrix Multiplication

- Given five matrices

A_1 A_2 A_3 A_4 A_5

- Which pair to multiply first?
- Plenty of choices…



A_1 A_2 A_3 A_4 A_5

A_1 A_2 * A_3 A_4 * A_5

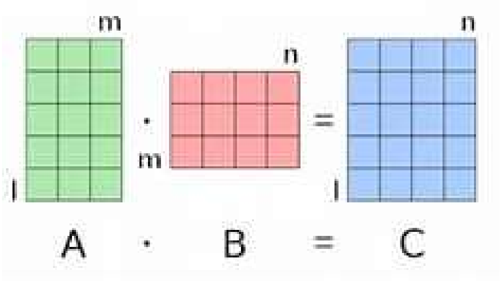
A_1 A_2 * A_3 * A_4 A_5

A_1 * A_2 A_3 * A_4 A_5

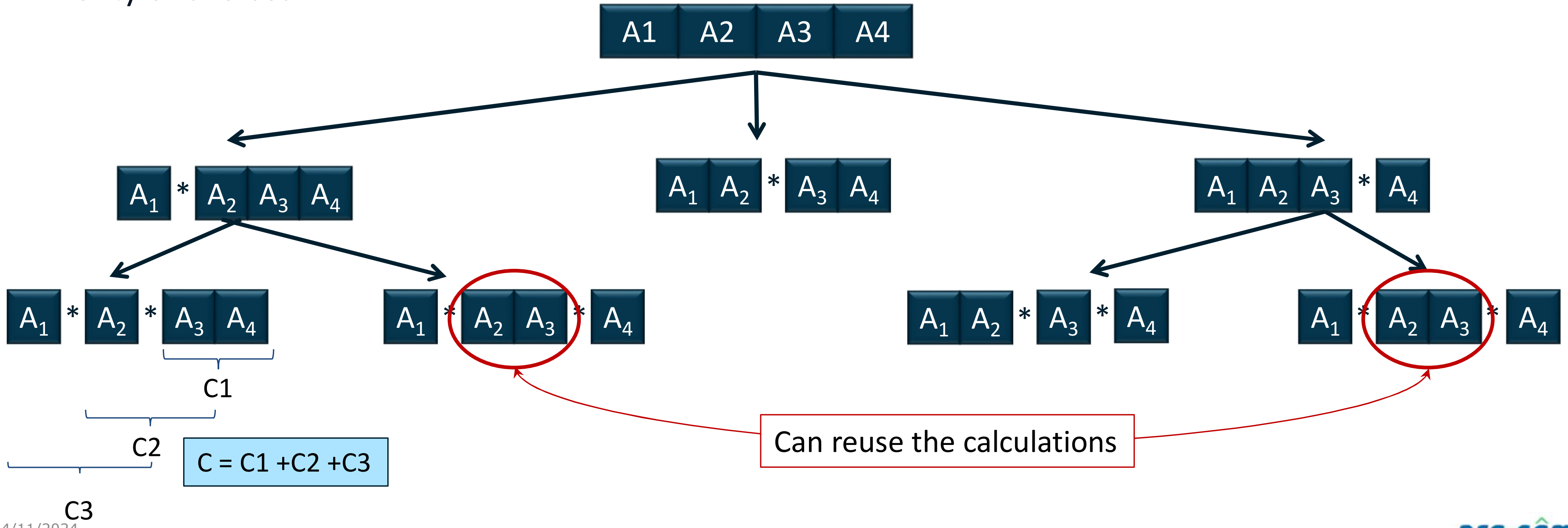
Example: Matrix Multiplication

- For sake of brevity, lets workout for four matrices

A_1 A_2 A_3 A_4



- Which pair to multiply first?
- Plenty of choices…


A1 A2 A3 A4

$A_1 * A_2 A_3 A_4$

$A_1 A_2 * A_3 A_4$

$A_1 A_2 A_3 * A_4$

$A_1 * A_2 * A_3 A_4$

$A_1 * A_2 A_3 * A_4$

$A_1 A_2 * A_3 * A_4$

$A_1 * A_2 A_3 * A_4$

C1

C2

C3

C = C1 +C2 +C3

Can reuse the calculations

So what is Combinatorial Optimization!

Basic Philosophy

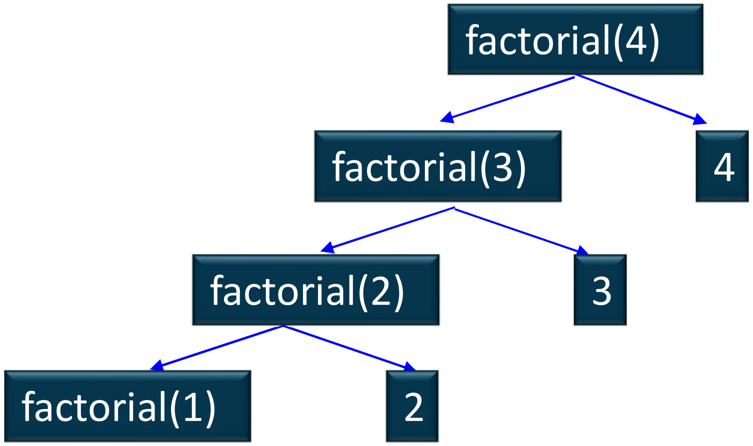
- A combinatorial optimization algorithm looks for the optimal solution without explicitly generating all possible solutions.
- Several Fundamental Approaches exist
- Iterative : Looking for common sub problems (Dynamic Programing)
  - Works well with problems having optimal substructure property
- Pruning : eliminate partial solutions based on solution found so far (branch and bound)
- Use heuristics to prioritize the explorations of partial solutions
  - Best first heuristic search
- Incremental iterative refinement approach (simulated annealing, genetic algorithms)

Dynamic Programming

Execution Trace (decomposition)

Factorial N

- ❑ Let’s take N = 4 for example



factorial(4)

factorial(3)

4

factorial(2)

3

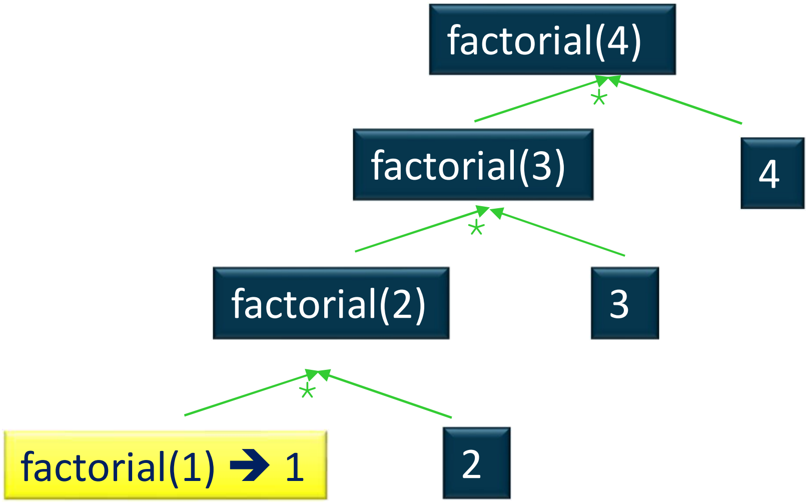
factorial(1)

2

Execution Trace (composition)

Factorial N

- ❑ Let’s take N = 4 for example



factorial(4)

*

factorial(3)

4

*

factorial(2)

3

*

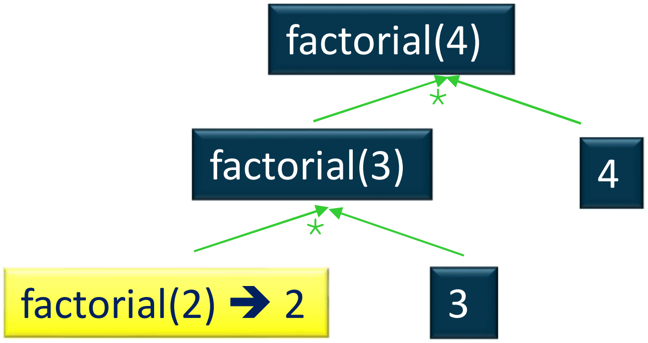
factorial(1) ➔ 1

2

Execution Trace (composition)

Factorial N

- ❑ Let’s take N = 4 for example



factorial(4)

*

factorial(3)

4

*

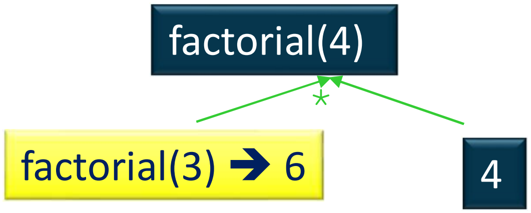
factorial(2) → 2

3

Execution Trace (composition)

Factorial N

❑ Let’s take N = 4 for example



Execution Trace (composition)

Factorial N

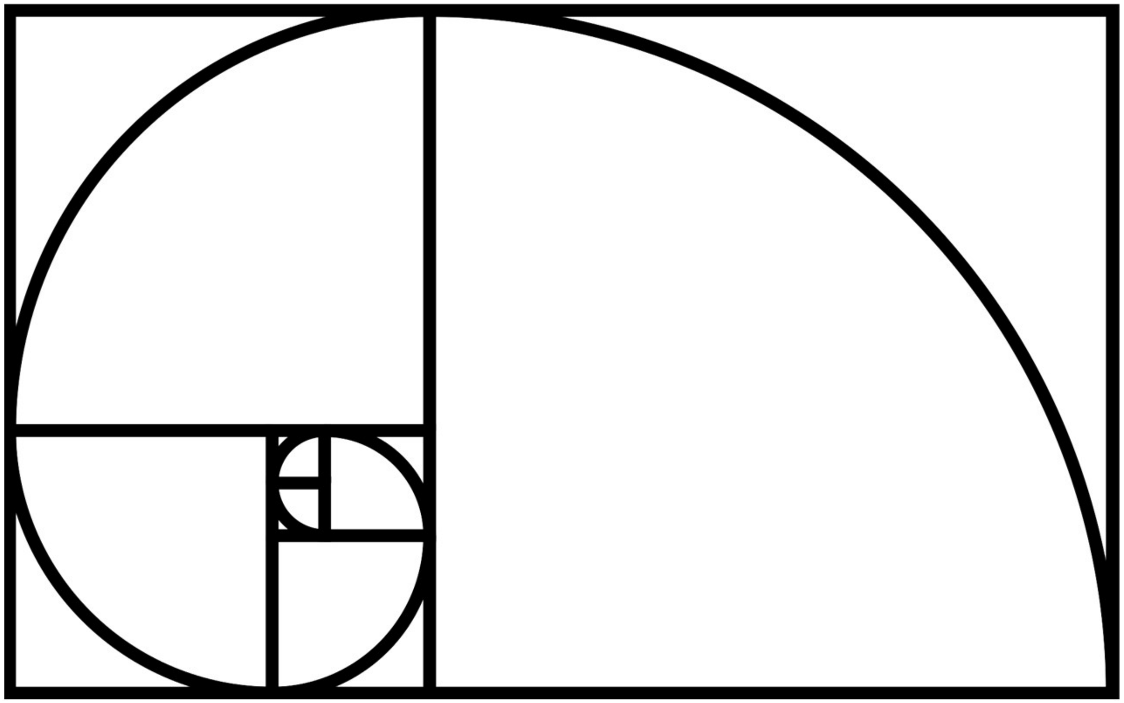
- ❑ Let’s take N = 4 for example

factorial(4) → 24

Fibonacci Numbers

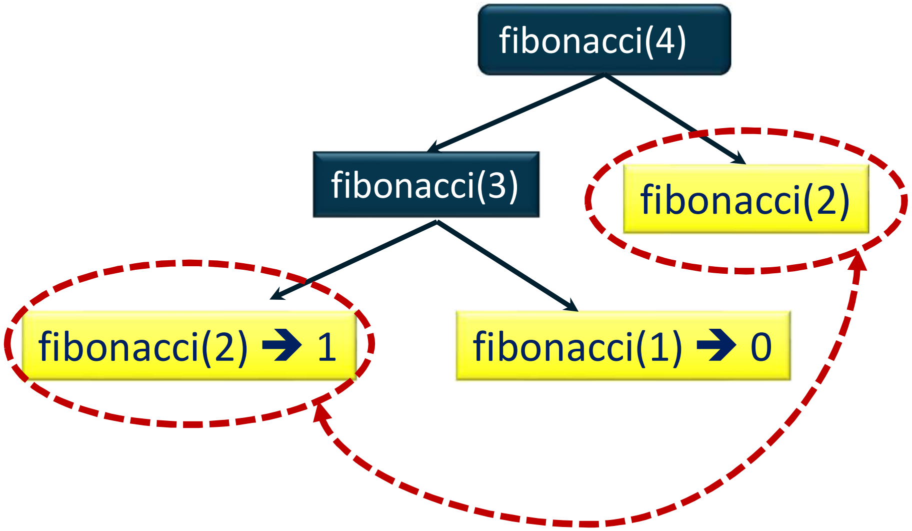
- The n^{th}Fibonacci number is the sum of the previous two Fibonacci numbers
  - 0, 1, 1, 2, 3, 5, 8, 13, …

- Recursive Design:
  - Decomposition & Composition
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - Base case:
    - fibonacci(1) = 0
    - fibonacci(2) = 1

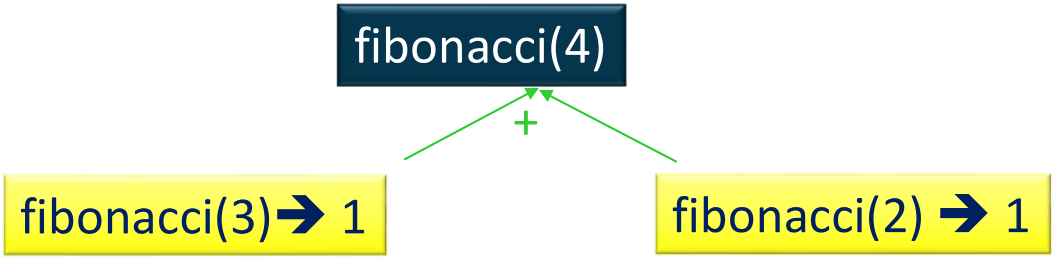


Execution Trace (decomposition)

- ❑ Again for N = 4



Execution Trace (composition)



fibonacci(4) ➔ 2

Key to Successful Dynamic Programming

- If-else statement
  - or some other branching statement
  - There should be an option to choose from
- Some branches: recursive call
  - “Smaller" arguments or solve "smaller" versions of the same task (decomposition)
  - Combine the results (composition) [if necessary]
- Other branches: no recursive calls
  - Stopping cases or base cases

Warning: Infinite Recursion May Cause a Stack Overflow Error

- Infinite Recursion
  - Problem not getting smaller (no/bad decomposition)
  - Base case exists, but not reachable (bad base case and/or decomposition)
  - No base case

- Stack: keeps track of recursive calls by OS
  - Method begins: add data onto the stack
  - Method ends: remove data from the stack

- Recursion never stops; stack eventually runs out of space
  - Stack overflow error

The 0-1 Knapsack Problem

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 8 kg of loot
  - He has to choose which of N items to steal
    - Each item has some weight and some value
    - “0-1” because each item is stolen (1) or not stolen (0)
  - He has to select the items to steal in order to maximize his loot
    - But cannot exceed 8 kg
- A greedy algorithm does not find an optimal solution
- A dynamic programming algorithm works finds an optimal solution
- In the 0-1 knapsack problem,
  - We can’t exceed the weight limit,
  - But the optimal solution may be less than the weight limit

The 0-1 Knapsack Explanation

Weight (kg)	1	2	3	5	6	8
Value (Rs.)	2	5	4	13	14	16

8 Kg

The 0-1 knapsack problem - Working

- ❑ It’s a subset sum problem
  - ❖ Find a subset S of A such that it maximizes $\sum_{a_j \in S} a_j$ under a constraint $\sum_{a_j \in S} a_j$ ≤ Q

- ❑ Recursive formulation:
  - ❖ Let cost (j, Q) be the cost of the optimal subset of "a_j“ within bounds of Q

$$\text{Cost (j, Q)} = \begin{cases} 0 & \dots\ for\ j = 0 \\ Cost(\ j-1,\ \ Q\ ) & \dots\ a_j > Q \\ \max\ \left[\ \underbrace{Cost(\ j-1, Q\ )}_{\text{Do not include } a_j \text{ in S}},\ \underbrace{a_j + Cost\big(j-1,\ \ Q\ -\ a_j\big)}_{\text{Include } a_j \text{ in S}}\ \right] & \dots otherwise \end{cases}$$

No elements, Nothing to include

Element a_j is larger than the capacity Q

Maximize the cost

How to identify?

- There must be recursion
  - Dynamic Programing is enhanced Recursion
- We should be able to split in overlapping sub-problems
  - Same calculations again and again
- Need to find an optimal solution
  - E.g. max, min, largest, smallest, slowest, fastest
- There must be choice:
  - More than one option to choose from

The Principle of Optimality

- ❑ Dynamic programming is a technique for finding an optimal solution
- ❑ The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all sub-problems
- ❑ The principle of optimality holds if
  - ❖ Every optimal solution to a problem contains...
  - ❖ ...optimal solutions to all sub-problems
- ❑ Example: Consider the problem of making N¢ with the fewest number of coins
  - ❖ Either there is an N¢ coin, or
  - ❖ The set of coins making up an optimal solution for N¢ can be divided into two nonempty subsets, n_1¢ and n_2¢
  - ❖ If either subset, n_1¢ or n_2¢, can be made with fewer coins, then clearly N¢ can't be made with fewer coins, hence solution was not optimal

The Principle of Optimality

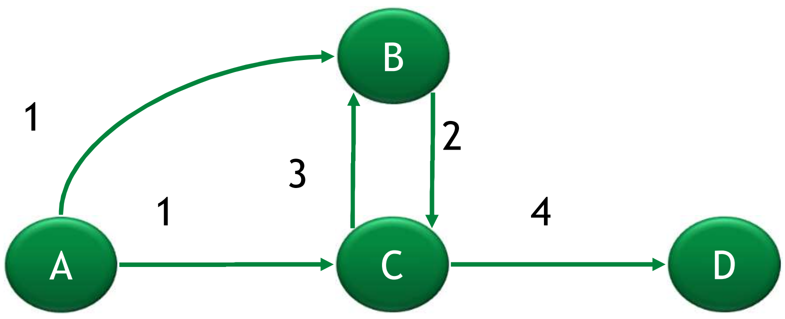
- The principle of optimality does not say
  - If you have optimal solutions to all sub-problems...
  - ...then you can combine them to get an optimal solution

- Example: In US coinage,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, and
  - The optimal solution to 6¢ is 5¢ + 1¢, but
  - The optimal solution to 13¢ is not 5¢ + 1¢ + 1¢ + 5¢ + 1¢

- But there is some way of dividing up 13¢ into subsets with optimal solutions (say, 10¢ + 1¢ + 1¢ + 1¢) that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

Longest Simple Path

- ❑ Consider the following graph:



- ❑ The longest simple path (path not containing a cycle) from A to D is A B C D
- ❑ However, the sub-path A-B is not the longest simple path from A to B (A-C-B is longer)
- ❑ The principle of optimality is not satisfied for this problem
- ❑ Hence, the longest simple path problem cannot be solved by a dynamic programming approach in this case

Comments

- Dynamic programming relies on working "from the bottom up" and saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential

Branch and Bound

Branch and Bound

- The basic concept underlying the branch-and-bound technique is to divide and conquer

- The basic steps are
  - Branching,
  - Bounding,
  - And fathoming (conquer)

- Since the original “large” problem is hard to solve directly,
  - It is divided into smaller and smaller sub-problems
  - Until these sub-problems can be conquered

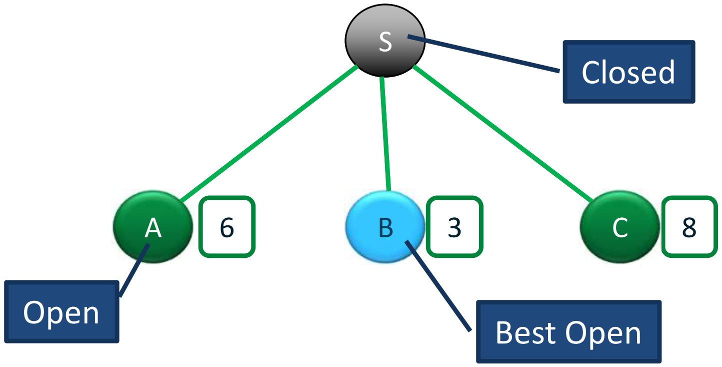
- The dividing (branching) is done by partitioning the entire set of feasible solutions into smaller and smaller subsets

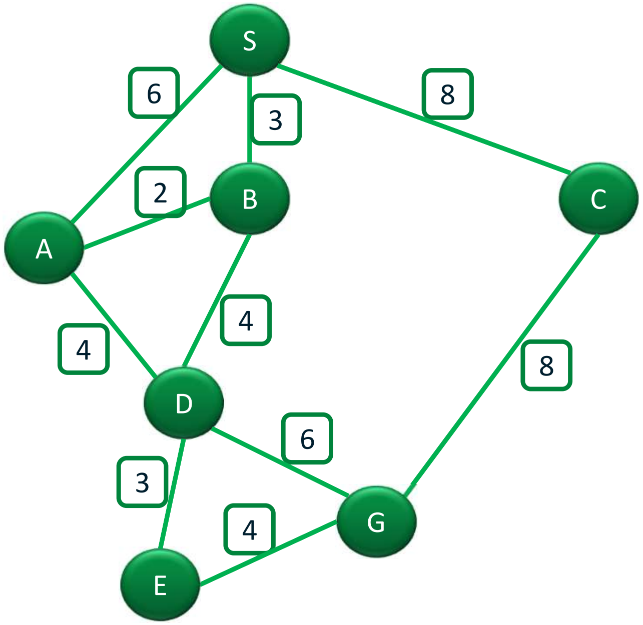
- The conquering (fathoming) is done partially by:
  - Giving a bound for the best solution in the subset;
  - Discarding the subset if the bound indicates that it can’t contain an optimal solution.

- Continue looking for solution until
  - A complete solution is found and
  - No other solution with smaller cost exists

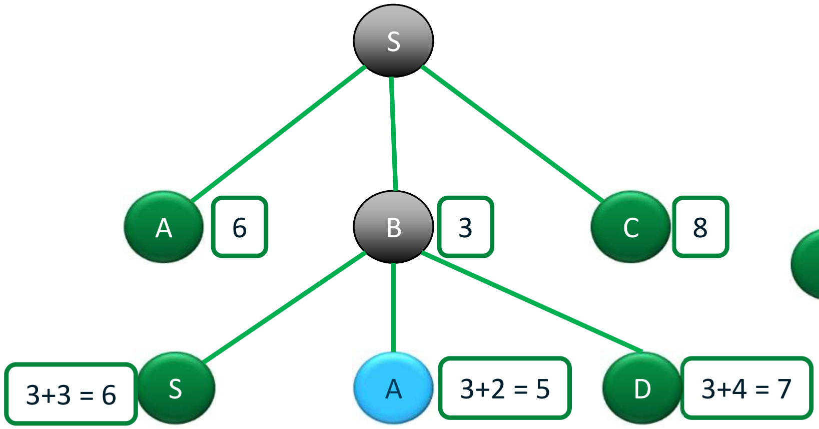
Prune those part of search space which cannot contain better solution!

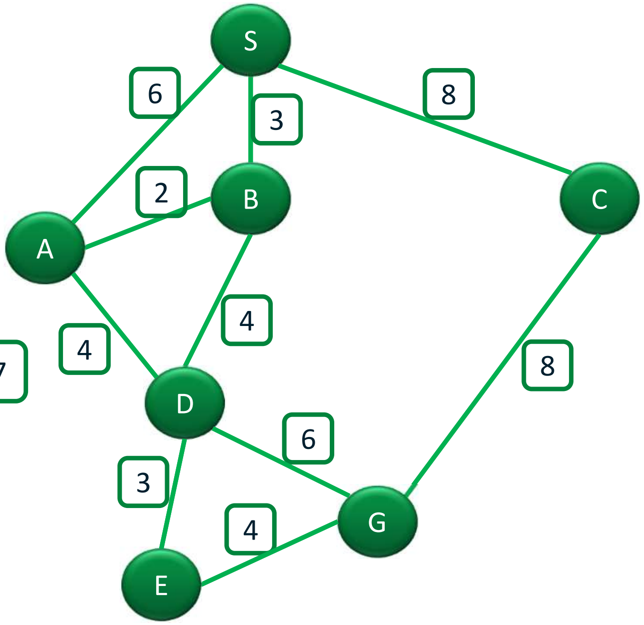
Example Problem with Edge Costs



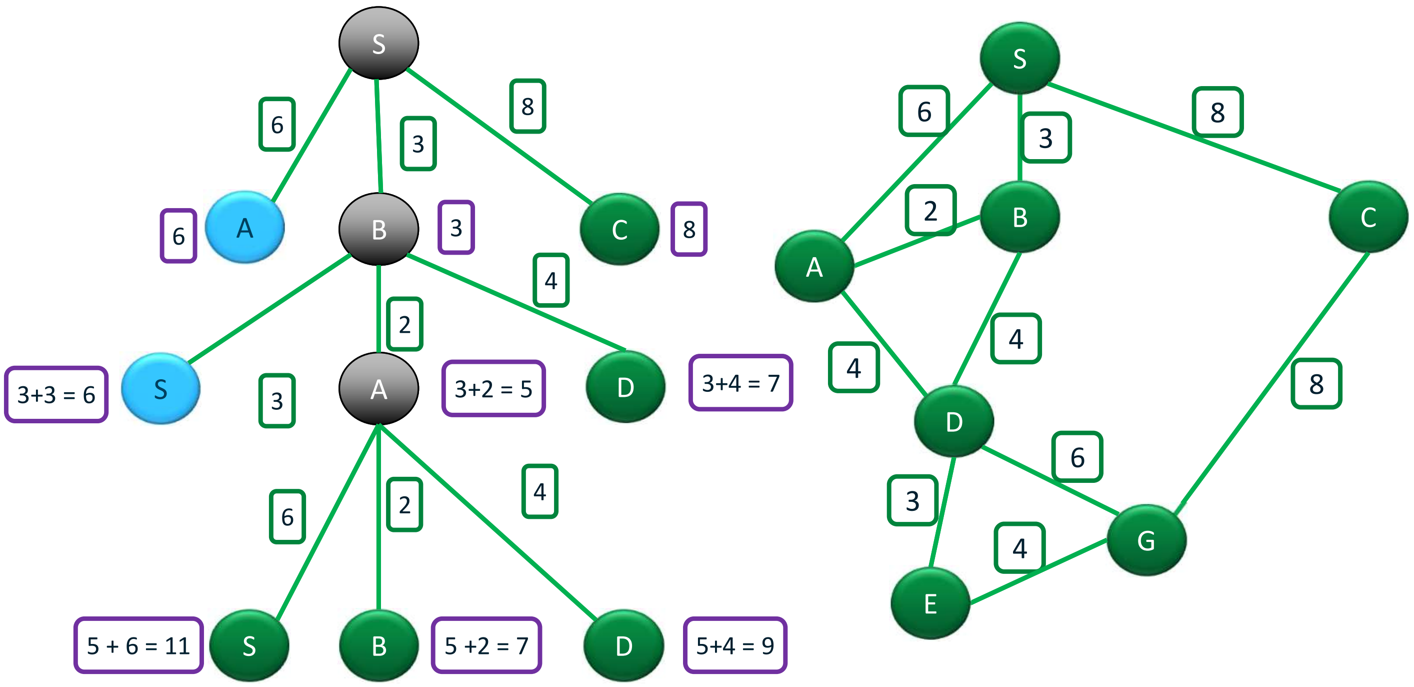


Example Problem with Edge Costs

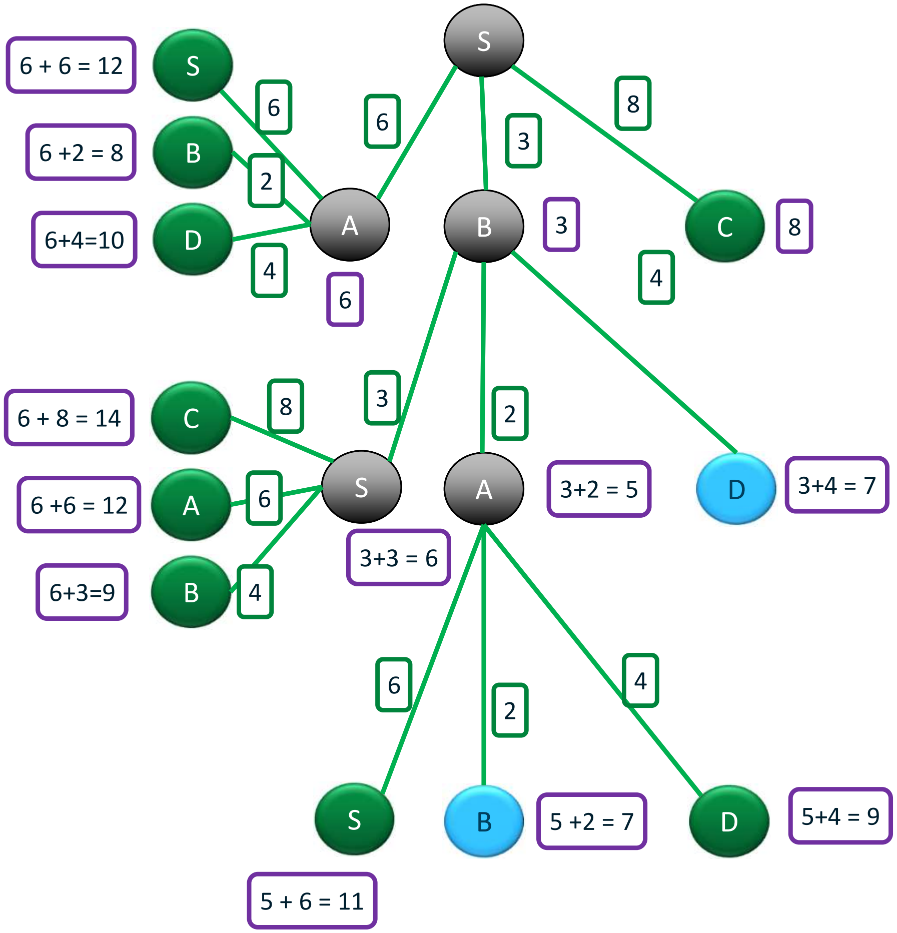


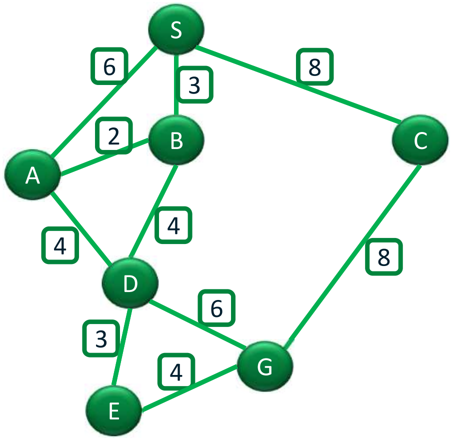


Example Problem with Edge Costs



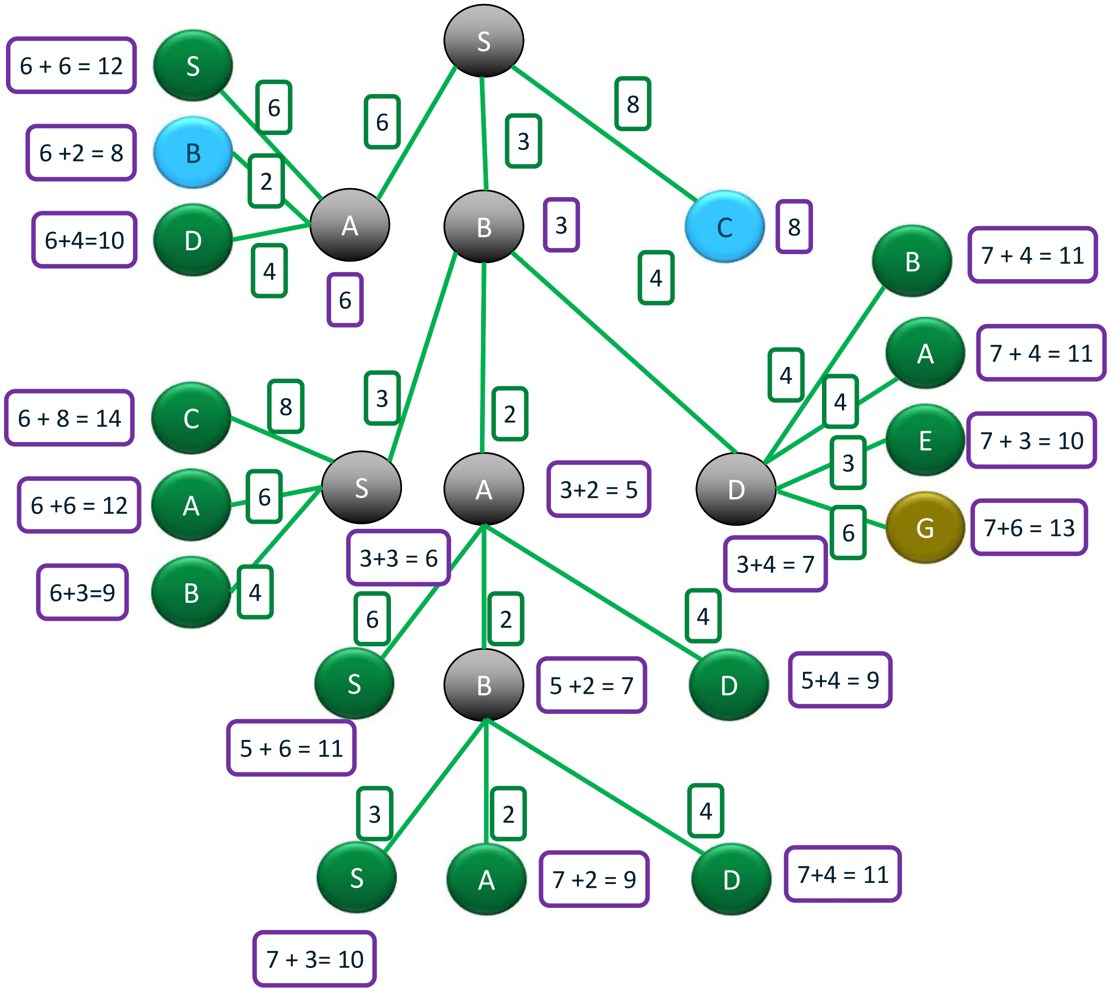
Example Problem with Edge Costs

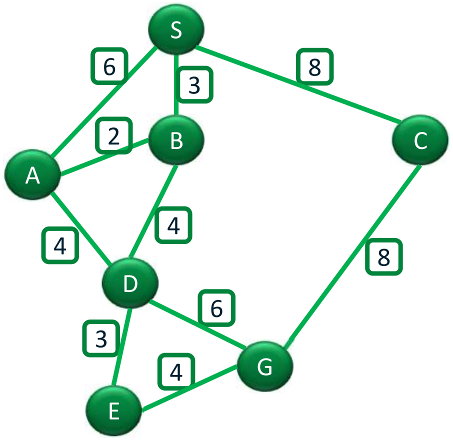




pra-sâmi

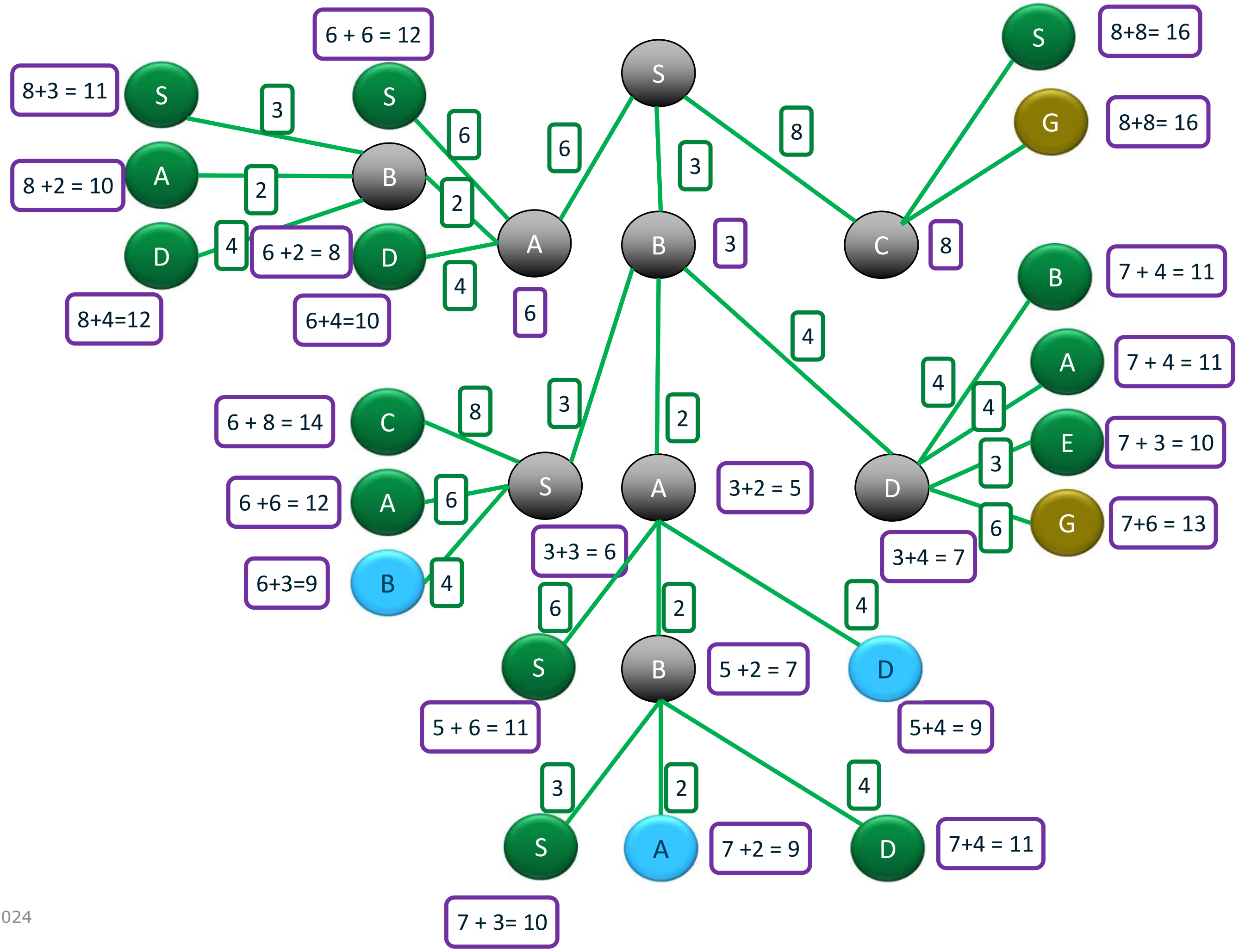
Example Problem with Edge Costs

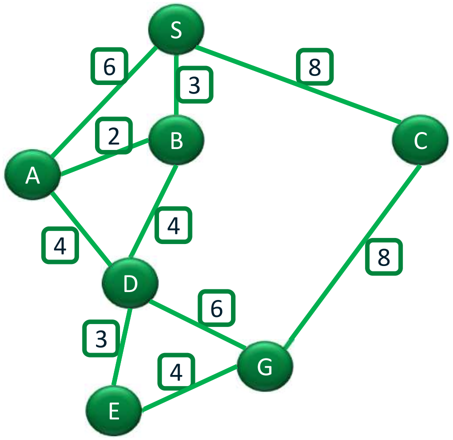




pra-sâmi

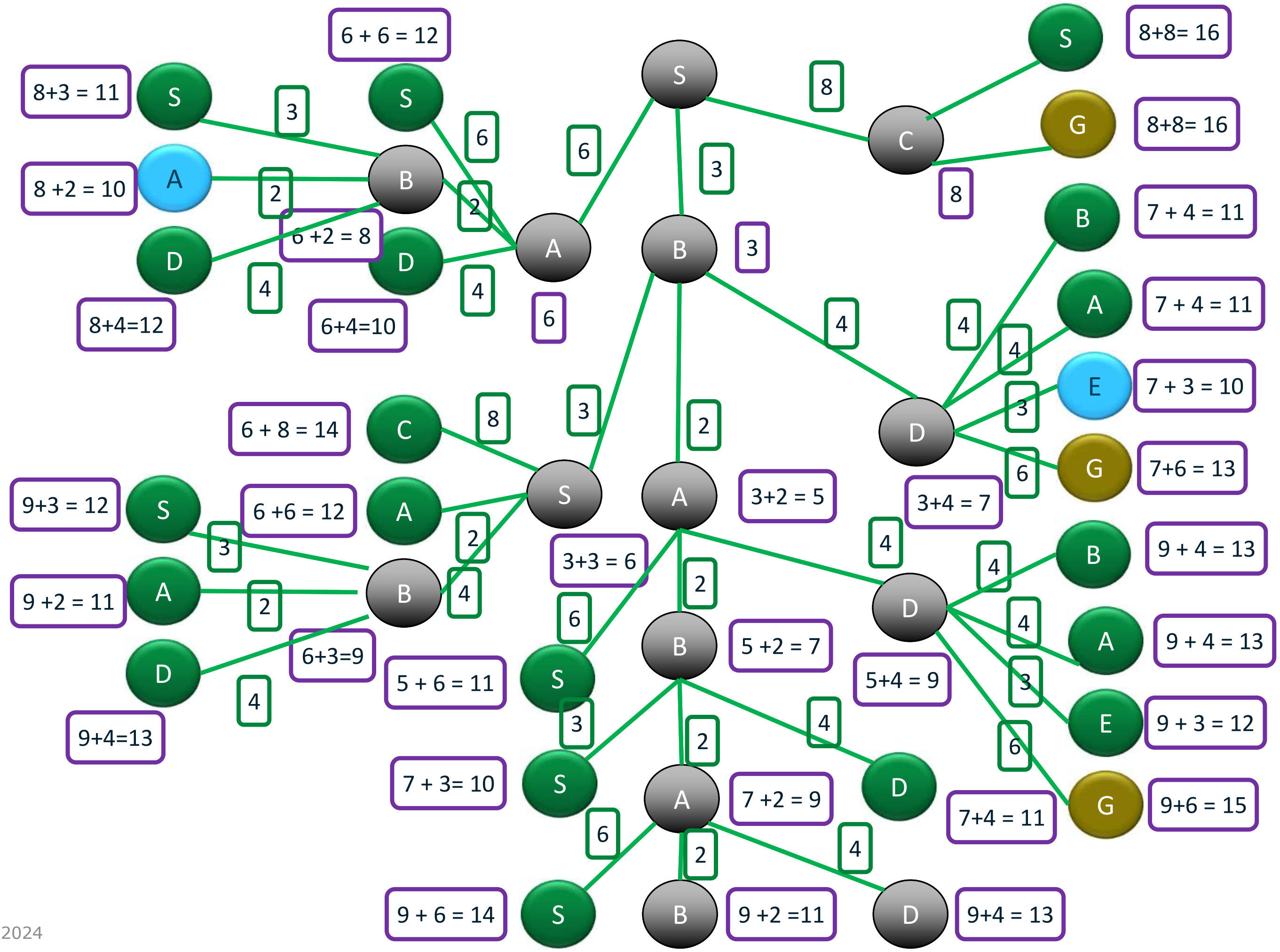
Example Problem with Edge Costs

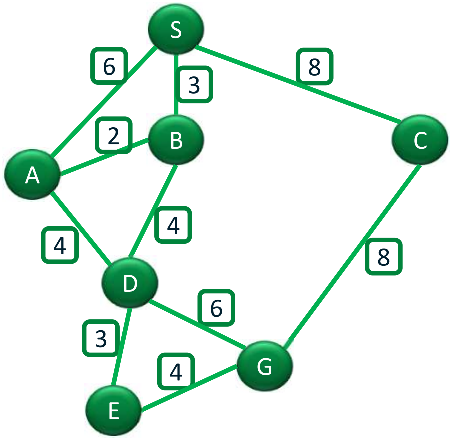




pra-sâmi

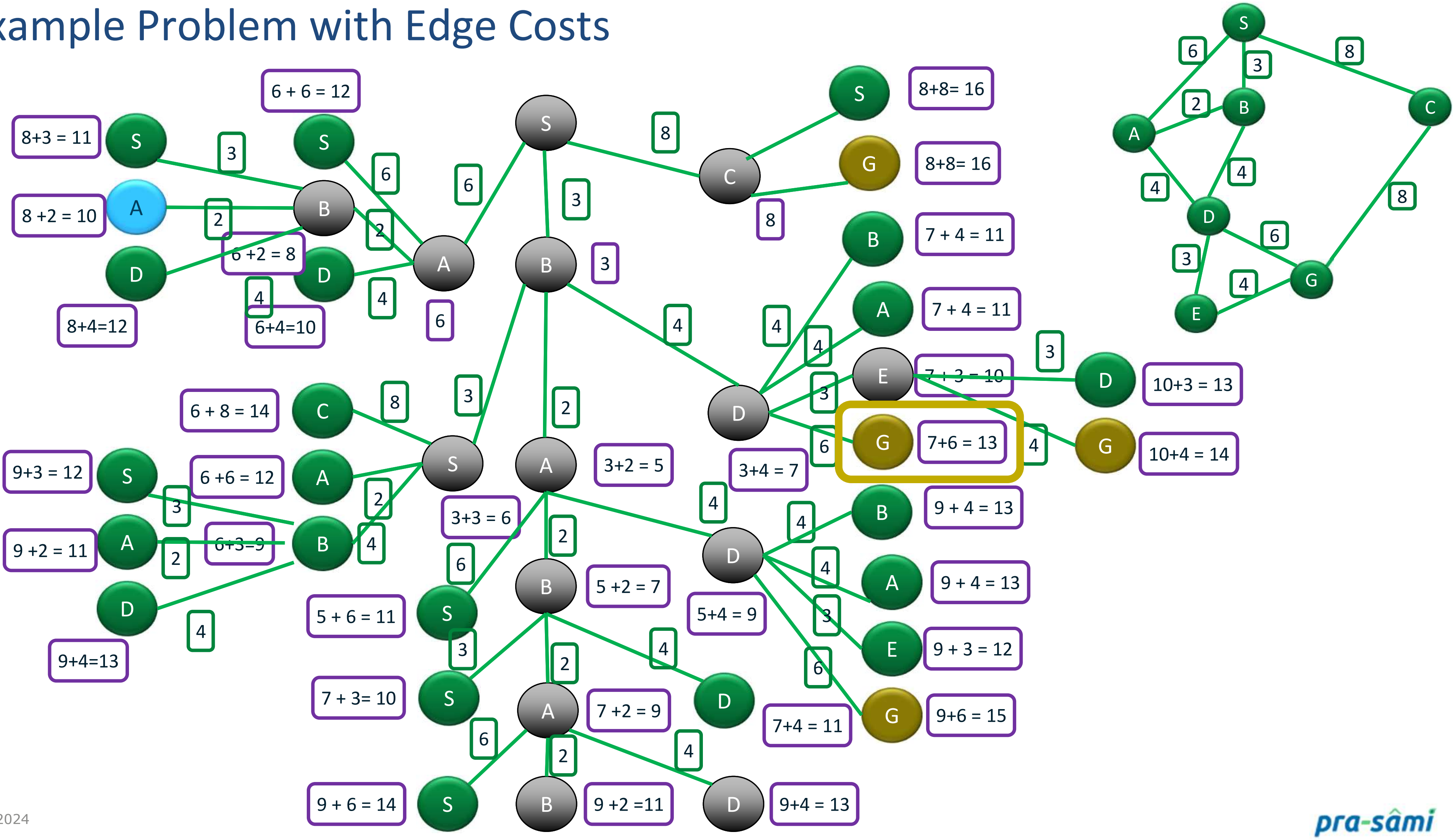
Example Problem with Edge Costs





pra-sâmi

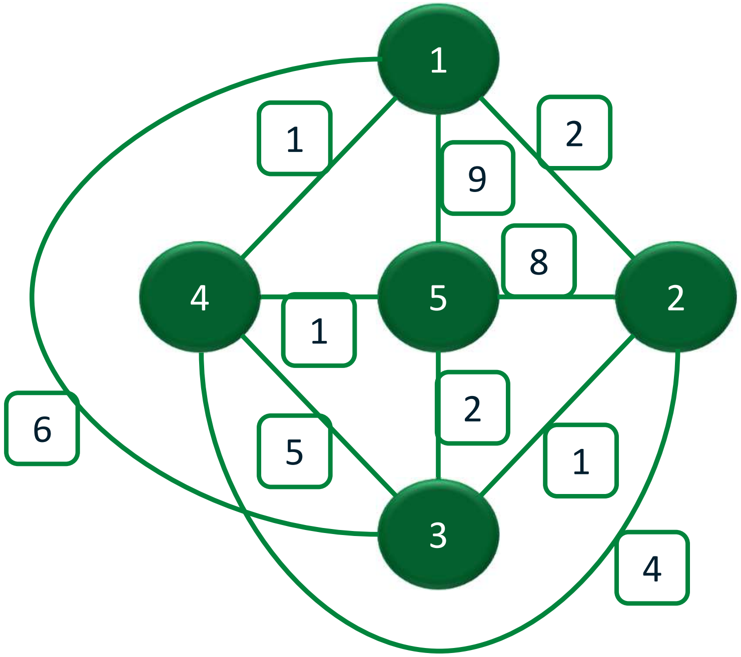
Example Problem with Edge Costs



Example Traveling Sales Person

- ❑ Given set of N cities and given the distances between them
- ❑ Sales person starts from a city, choice does not matter
- ❑ Goes to each city once and once only
  - ❖ Cannot revisit a city
- ❑ Cost of travelling is proportional to distance between the cities
- ❑ Find a Hamiltonian Cycle having least cost
  - ❖ Find a tour so that total distance travelled in lowest

- ❑ Other Examples : stocking robot at an assembly line; Newspaper Boy

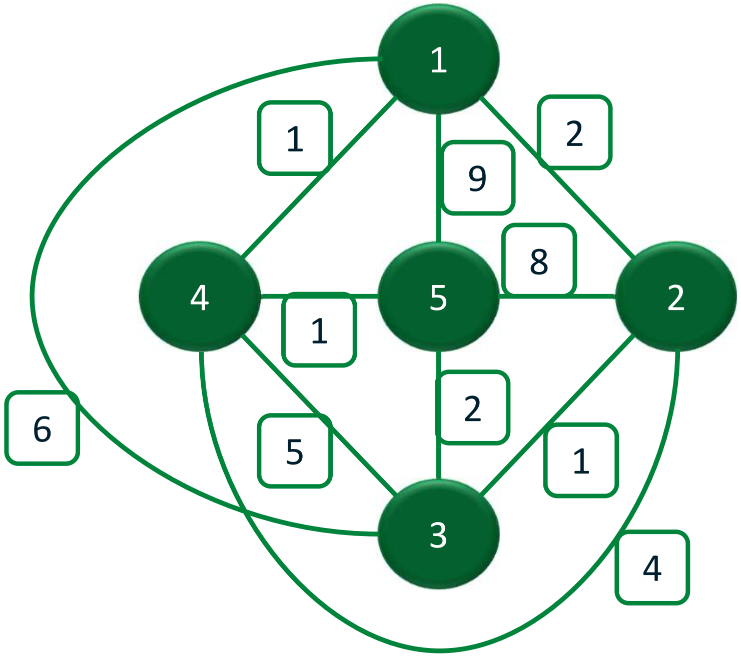


Example Traveling Sales Person

- ❑ MoveGen Function
  - ❖ 1 → (2, 4, 5, 3)
  - ❖ 2 → (1, 3, 4, 5)
  - ❖ 3 → (1, 2, 4, 5)
  - ❖ 4 → (1, 2, 3, 5)
  - ❖ 5 → (1, 2, 3, 4)

- ❑ Cost of path is sum of edges in the path

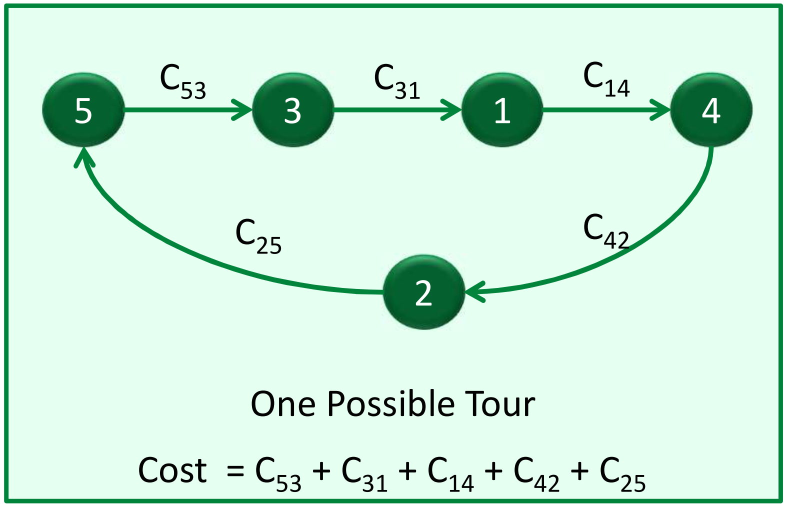
- ❑ State Space



Example Traveling Sales Person

- How many tours possible?
  - Start from any city (does not really matter
  - Number of remaining hops are permutation of remaining city
  - Number of tours = (n-1)!
    - Number of cyclic permutation
    - Tour 5-3-1-4-2-5 is same as tour 3-1-4-2-5-3

- State space for the problem will be any permutation of the cities

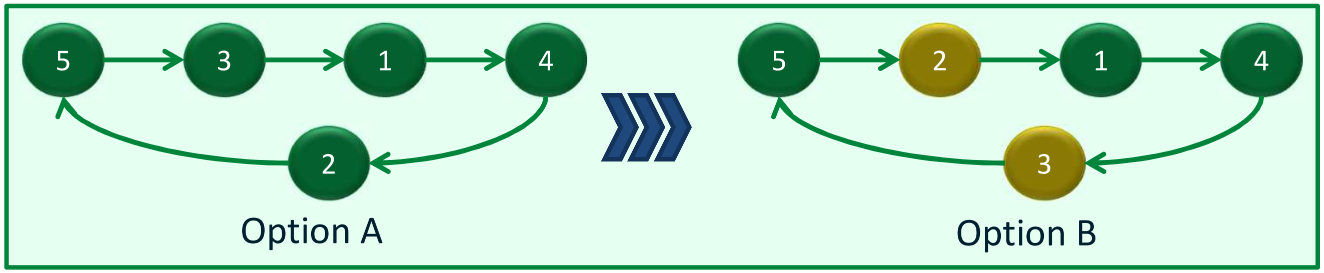


One Possible Tour

Cost = $C_{53} + C_{31} + C_{14} + C_{42} + C_{25}$

Approaches

- ❑ Two ways to find the solution
  - ❖ Iterative approach
  - ❖ Constructive Approach



- ❑ Iterative Approach
  - ❖ Start with one combination and change it to move towards better approach

- ❑ Constructive Approach

Iterative Improvement

- Find a subset S of A such that it maximizes $\sum_{a_j \in S} a_j$ under a constraint

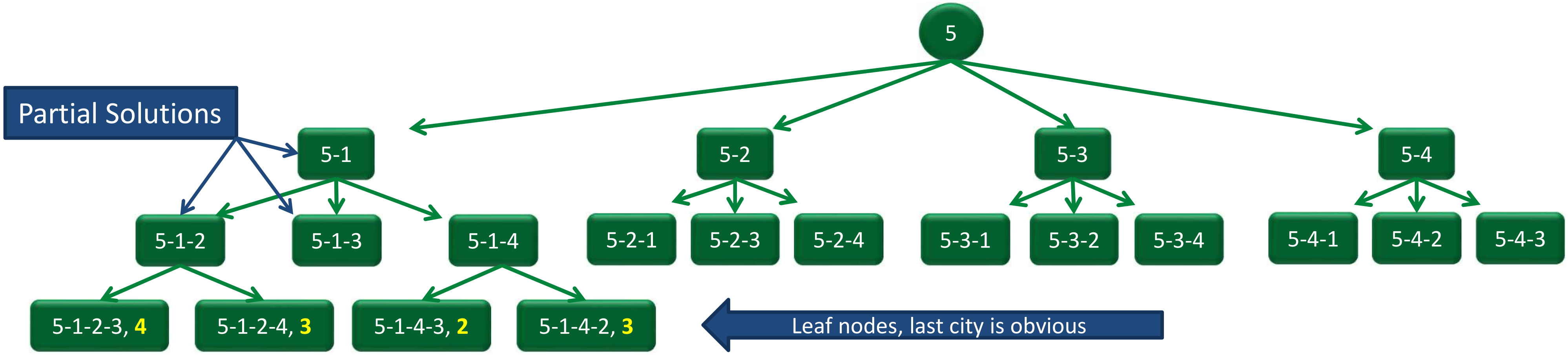
  $\sum_{a_j \in S} a_j \leq$ Q

- General Idea
  - Start with any maximal subset Z of A
  - Consider a pair (a_j , a_k) such that $a_j \in Z$ and $a_k \notin Z$.
  - Replace a_j with a_k in Z in subset sum increases but does not exceed Q.

- Such methods are called hill-climbing algorithms / gradient descent

Approaches

- ❑ Constructive Approach

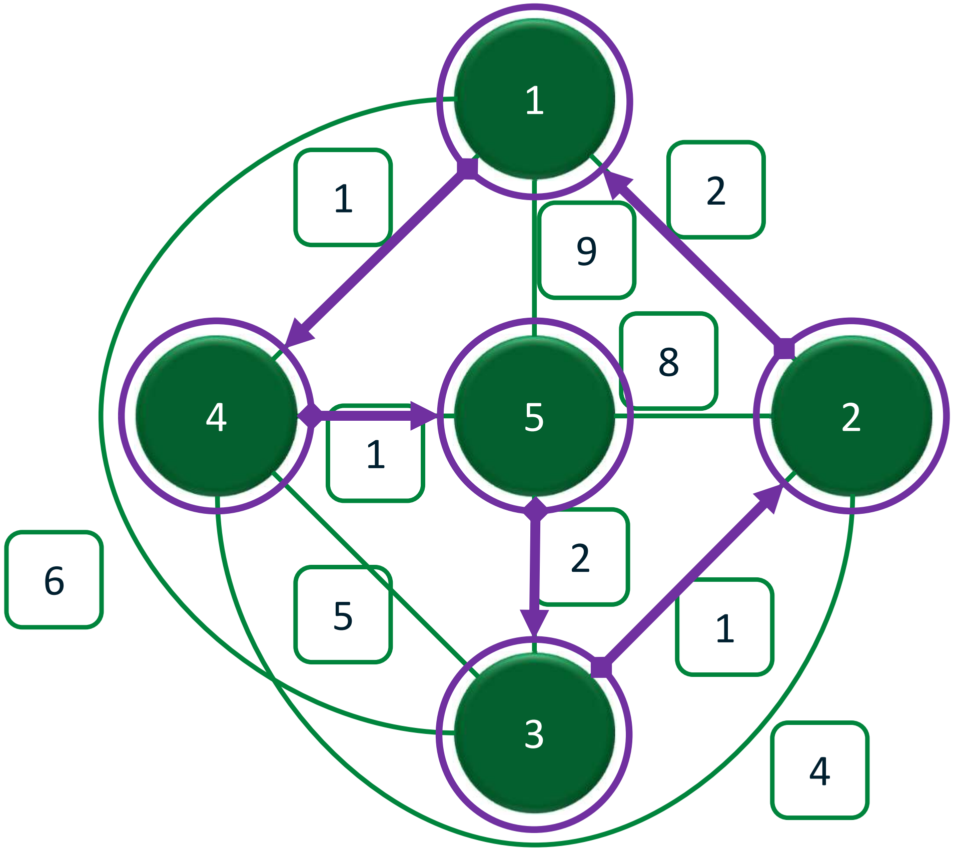


Partial Solutions

Leaf nodes, last city is obvious

Greedy Constructive Method

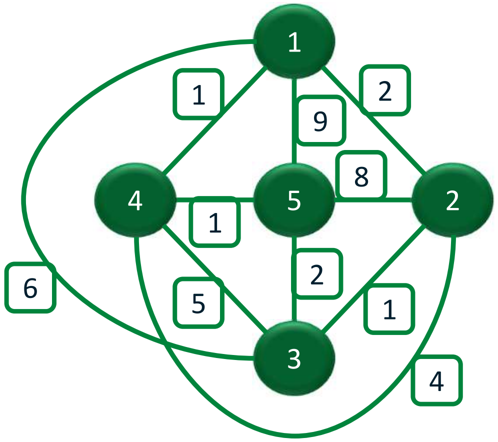
- ❑ Nearest Neighbor Heuristic
  - ❖ Start at some city
  - ❖ Move to nearest neighbor
  - ❖ Make sure to not to close the loop prematurely
- ❑ Route Total = 1 + 1 + 2 + 1 + 2 = 7
- ❑ Works in our case
  - ❖ May not work in all cases!

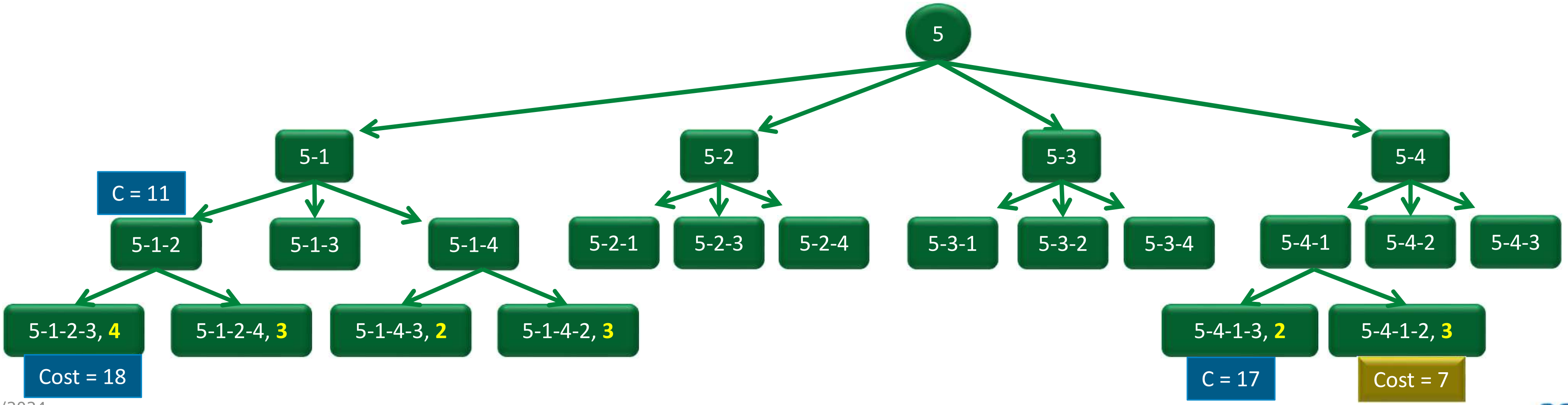


pra-sâmi

Heuristic Estimates to Help in Pruning

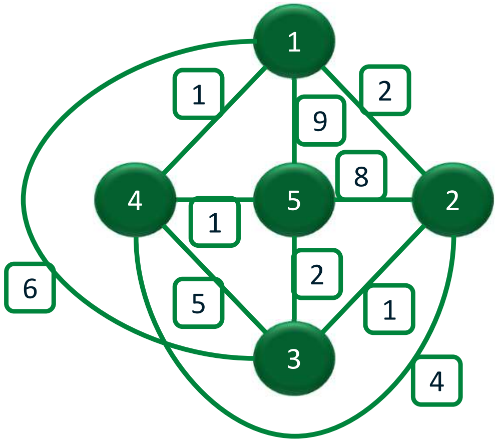
- Permutation: 5-4-1-2-3-5
  - Cost = 1+1+2+1+2 = 7
- At Every node, heuristic estimates can be kept as a guideline to prune
- In travelling salesman problem, heuristic estimates can be taken as average distance of remaining nodes multiplied by remaining nodes
- Use of heuristic can really help bring complexity of problem down





Branch and Bound

- ❑ Based on constructive approach
- ❑ Permutation: 1-2-3-4-5
  - ❖ Cost = 2+1+5+1+9 = 18
- ❑ Permutation: 1-2-3-5-4
  - ❖ Cost = 2+1+2+1+1 = 7
- ❑ As can be seen that above permutation is lowest and no other solution will come close
  - ❖ Hence we can prune other branches of the tree
- ❑ It is a back-tracking approach

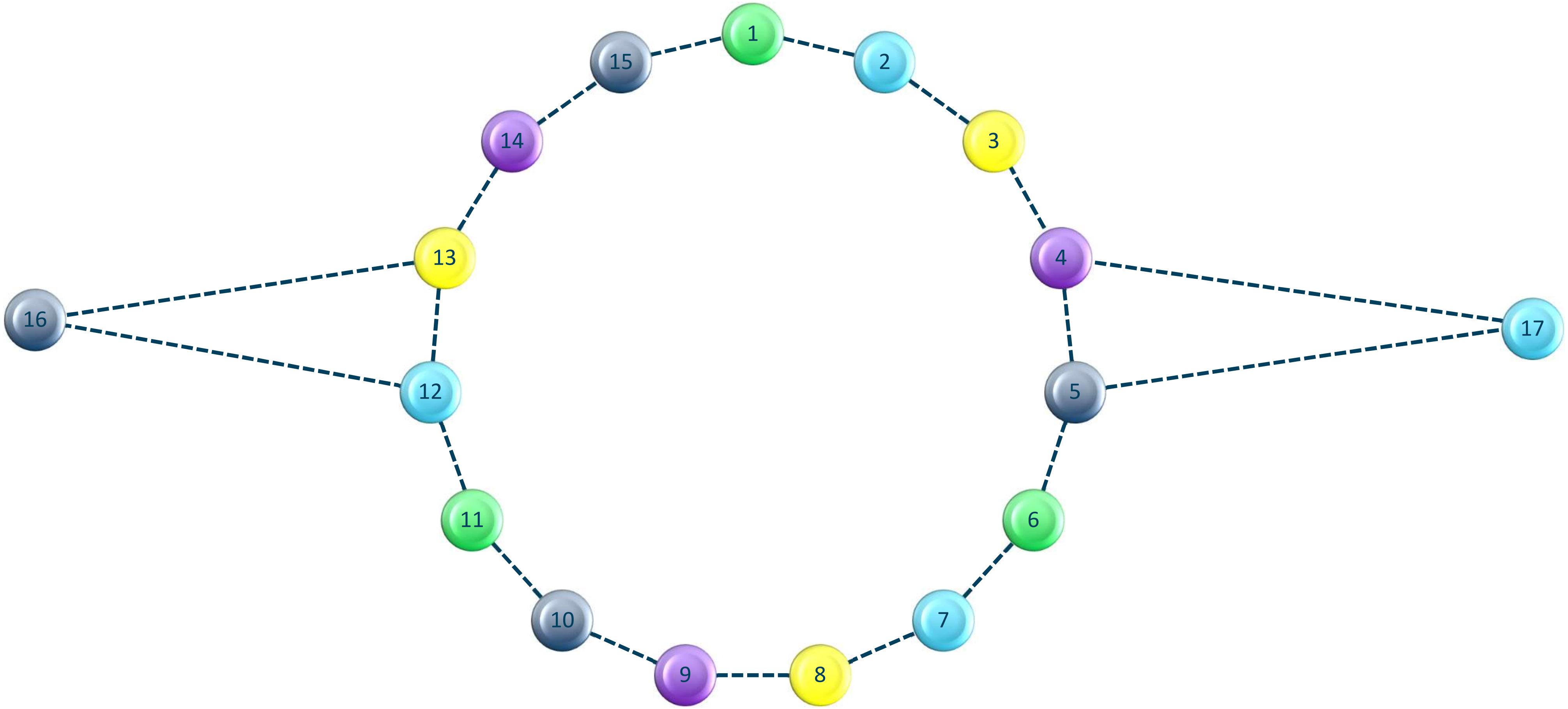


Cities	1	2	3	4	5
1	0	2	6	1	9
2		0	1	4	8
3			0	5	2
4				0	1
5					0

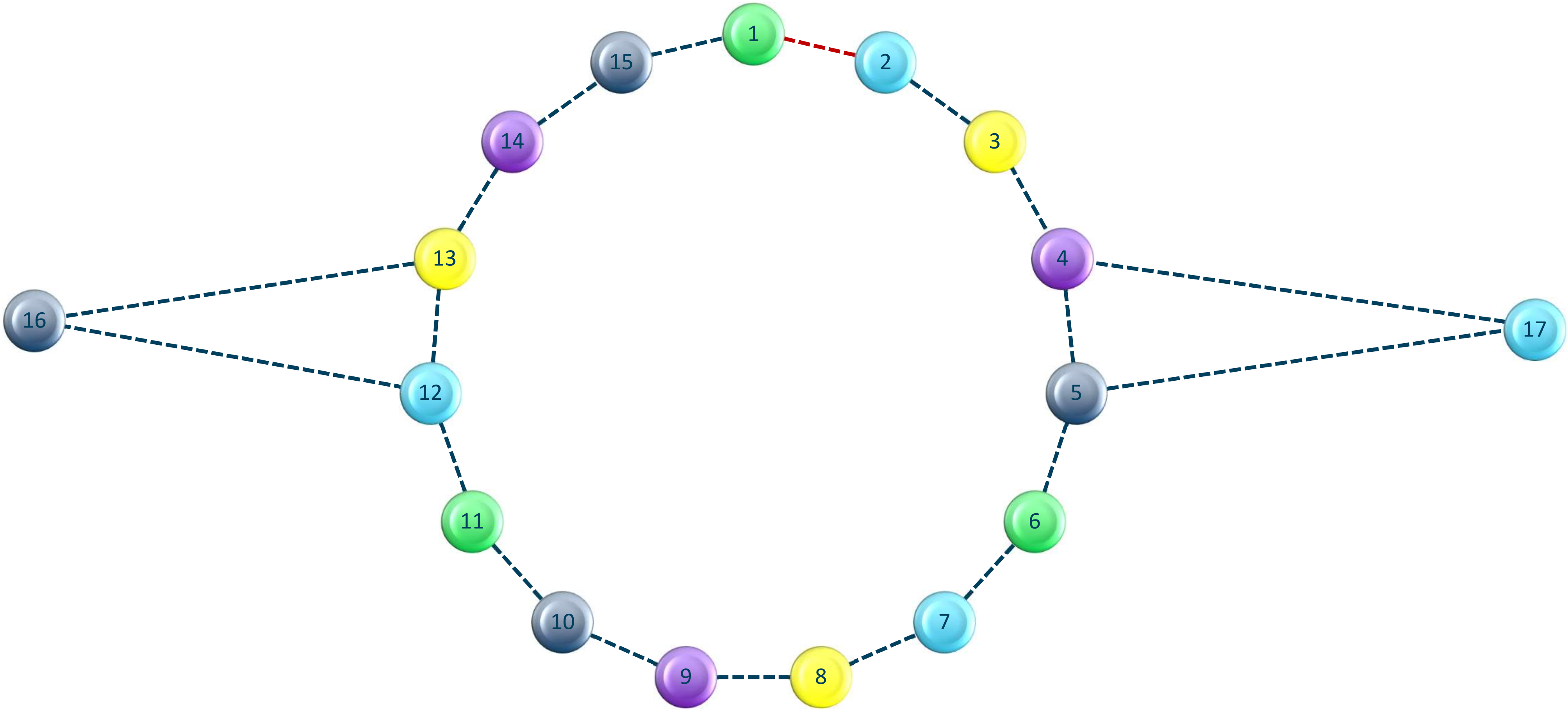
TSP : Greedy Approaches

- Nearest Neighbor
- Nearest Neighbor with a variation
  - Extend the partial tour at either end
- Greedy Heuristic
  - Sort the edges
  - Add shortest available edge to the tour

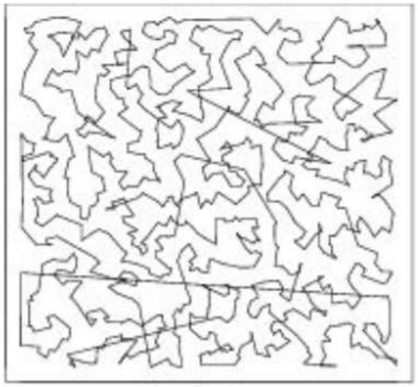
Example where it may not work



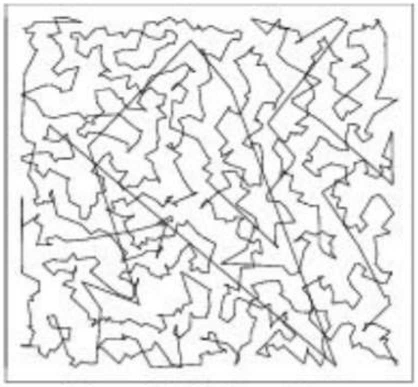
Example where it may not work



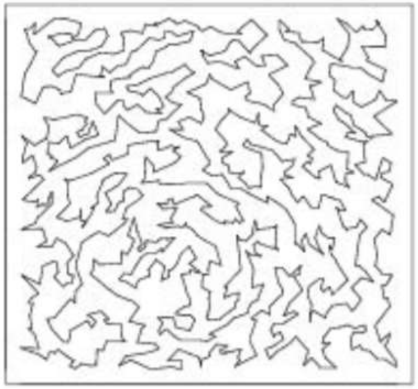
The Constructive Algorithms in Action



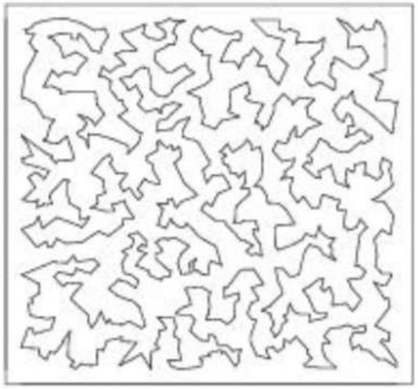
Greedy Tour



Nearest Neighbor Tour



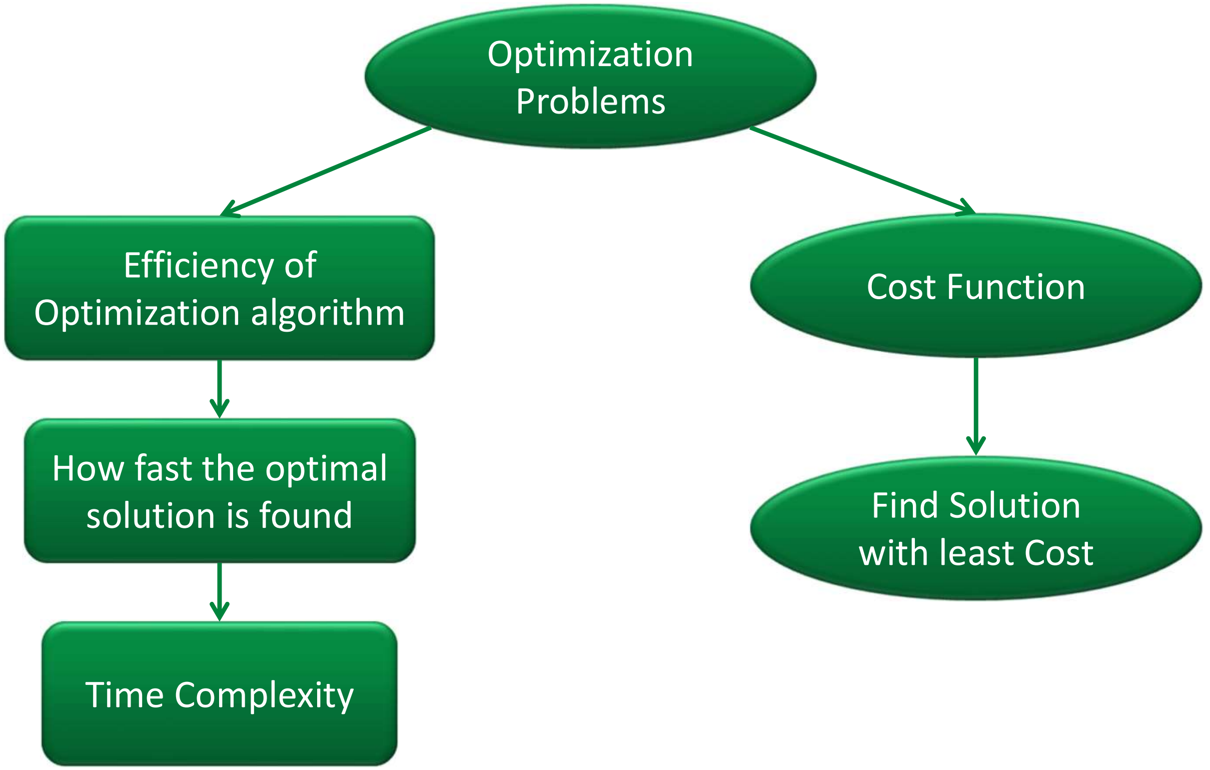
Saving Tour



Optimal Tour

Tours found by some heuristic constructive algorithms
Acknowledgement :
www.research.att.com/~dsj/chtsp

Cost and Complexity


Optimization Problems
Efficiency of Optimization algorithm
How fast the optimal solution is found
Time Complexity
Cost Function
Find Solution with least Cost

Experts are still designing algorithms which are optimal and faster…
Stay tuned as it is a moving target

THANK YOU

EXTRA MATERIAL

Dynamic Programing

Counting Coins

- Coins Available
  - 25, 10, 5, 1
  - 21,
- Amount to pay
  - 31
  - 63

- To find the minimum number of US coins to make any amount (31 or 63 ¢) , the greedy method always works (?)
  - At each step, just choose the largest coin that does not overshoot the desired amount: 31¢ = 25 + 5 +1
- The greedy method would not work if we did not have 5¢ coins
  - For 31 cents, the greedy method gives seven coins (25 + 1 + 1 + 1 + 1 + 1 + 1),
  - But we can do it with four (10 + 10 + 10 + 1)
- For 63 cents, the greedy method also would not work if we had a 21¢ coin
  - the greedy method gives six coins (25 + 25 + 10 + 1 + 1 + 1), but we can do it with three (21 + 21 + 21)
- How can we find the minimum number of coins for any given coin set?

Coin set for examples

- For the following examples, we will assume coins in the following denominations:
  1¢ 5¢ 10¢ 21¢ 25¢
- We’ll use 22¢ and 63¢ as our goal

A Simple Solution

- We always need a 1¢ coin, otherwise no solution exists for making one cent
  - Base condition considering our target
- To make K cents:
  - If there is a K-cent coin, then that one coin is the minimum
  - Otherwise, for each value i < K,
    - Find the minimum number of coins needed to make i cents
    - Find the minimum number of coins needed to make K - i cents
  - Choose the i that minimizes this sum
- This algorithm can be viewed as divide-and-conquer, or as brute force
  - This solution is very recursive
  - Compute time increases exponentially

Another Solution

- We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left
- For 63¢:
  - One 1¢ coin plus the best solution for 62¢
  - One 5¢ coin plus the best solution for 58¢
  - One 10¢ coin plus the best solution for 53¢
  - One 21¢ coin plus the best solution for 42¢
  - One 25¢ coin plus the best solution for 38¢
- Choose the best solution from among the 5 given above
- Instead of solving 62 recursive problems, we solve 5
- This is still a very expensive algorithm

A dynamic programming solution

- Idea: Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
    - Save each answer in an array !
- For each new amount N, compute all the possible pairs of previous answers which sum to N
    - For example, to find the solution for 13¢,
        - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
        - Next, choose the best solution among:
            - Solution for 1¢ + solution for 12¢
            - Solution for 2¢ + solution for 11¢
            - Solution for 3¢ + solution for 10¢
            - Solution for 4¢ + solution for 9¢
            - Solution for 5¢ + solution for 8¢
            - Solution for 6¢ + solution for 7¢

Example

- Suppose coins are 1¢, 3¢, and 4¢
  - There’s only one way to make 1¢ (one coin)
  - To make 2¢, try 1¢+1¢ (one coin + one coin = 2 coins)
  - To make 3¢, just use the 3¢ coin (one coin)
  - To make 4¢, just use the 4¢ coin (one coin)
  - To make 5¢, try
    - 1¢ + 4¢ (1 coin + 1 coin = 2 coins)
    - 2¢ + 3¢ (2 coins + 1 coin = 3 coins)
    - The first solution is better, so best solution is 2 coins
  - To make 6¢, try
    - 1¢ + 5¢ (1 coin + 2 coins = 3 coins)
    - 2¢ + 4¢ (2 coins + 1 coin = 3 coins)
    - 3¢ + 3¢ (1 coin + 1 coin = 2 coins) – best solution
  - Etc.