Find your way here

CSC 250: Foundations of Computer Science I

Fall 2023 - Lecture 4

Amitabha Dey

Department of Computer Science
University of North Carolina at Greensboro

Brute Force or Exhaustive Search

- Many problems can be thought of as search problems. A search problem involves some set of possibilities and we are looking for one or more of the possibilities that satisfy some property. Examples:
  - Searching for the shortest path from SF to NYC
  - Searching for the best combination of items to make in a factory
- For some problems we need to understand and perhaps generate the set of all possibilities over which we will search.
- Method for brute force or exhaustive search of all possibilities:
  - Generate a list of all potential solutions to the problem
  - Evaluate potential solutions one by one disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far.

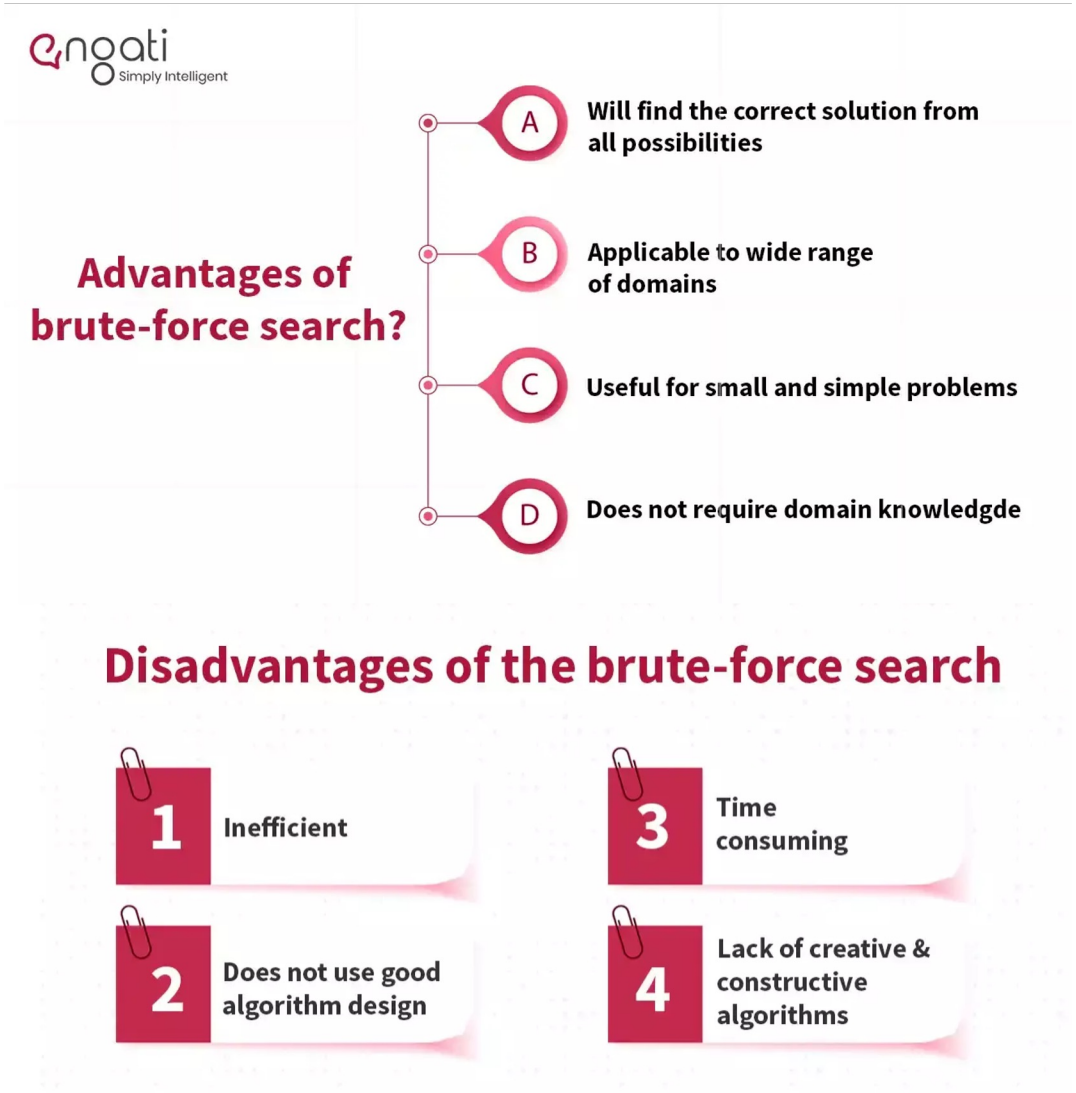


Image credit - https://www.engati.com/

Brute Force or Exhaustive Search

Unlocking 4-digit Padlock

For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock.
Since you can't remember any of the digits, you have to use a brute force method to open the lock.
So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take 10^4, or 10,000 tries to find your combination.

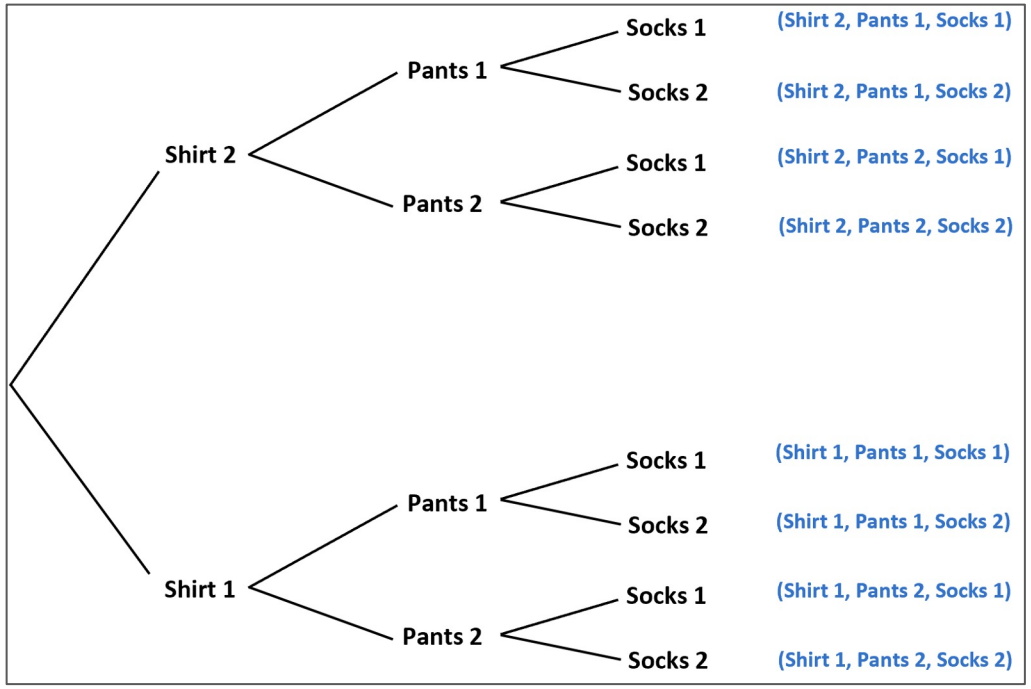


Image credit - https://www.statology.org/

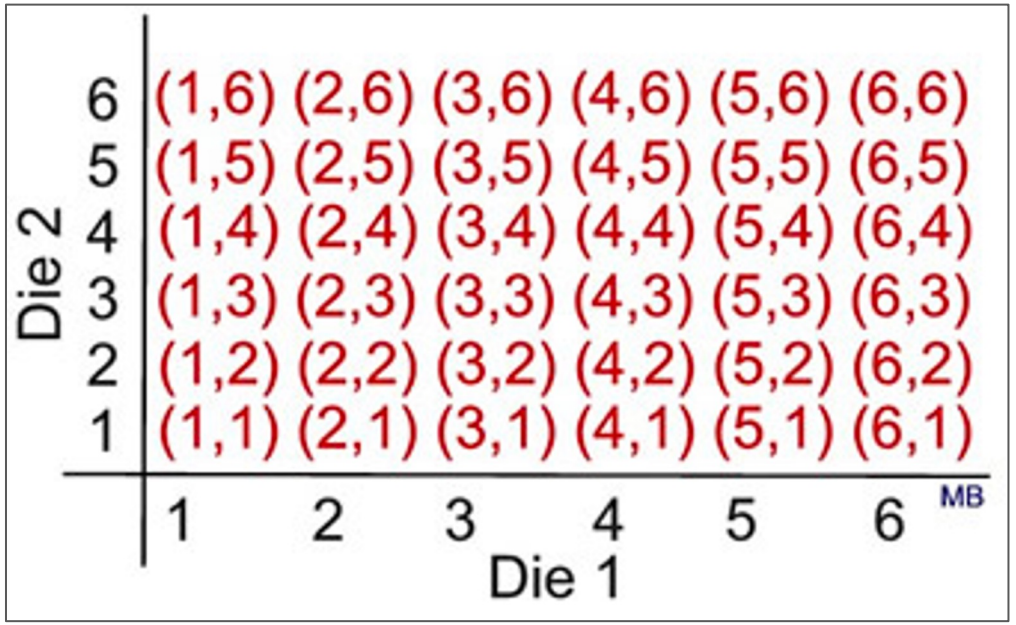


Image credit - https://mathbitsnotebook.com/

Brute Force or Exhaustive Search

Eight Queens Puzzle

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

There are 92 solutions. The problem was first posed in the mid-19th century. In the modern era, it is often used as an example problem for various computer programming techniques.

The eight queens puzzle is a special case of the more general n queens problem of placing n non-attacking queens on an n×n chessboard. Solutions exist for all natural numbers n with the exception of n = 2 and n = 3.

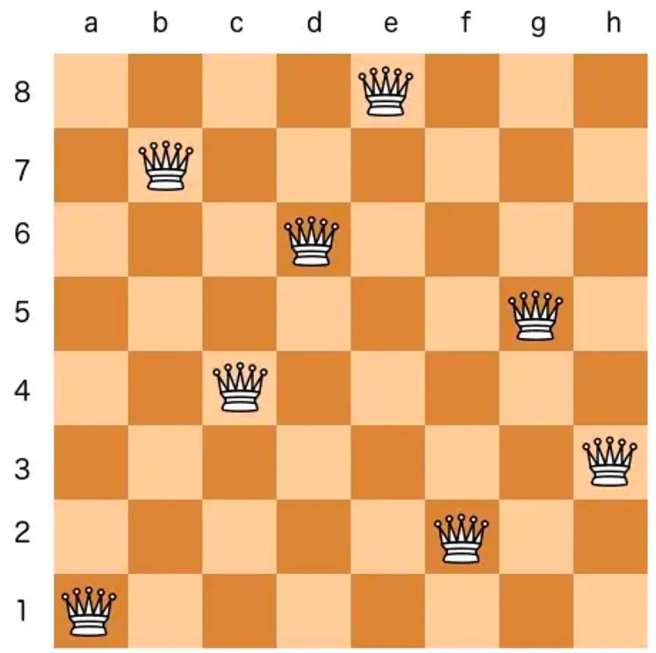


Image credit - Colin M.L. Burnett

The problem of finding all solutions to the 8-queens problem can be quite computationally expensive, as there are **4,426,165,368** possible arrangements of eight queens on an 8×8 board, but only **92** solutions.

It is possible to use shortcuts that reduce computational requirements or rules of thumb that avoids brute-force computational techniques.

For example, by applying a simple rule that chooses one queen from each column, it is possible to reduce the number of possibilities to 16,777,216 (that is, 8^8) possible combinations.

Brute Force or Exhaustive Search

Eight Queens Puzzle

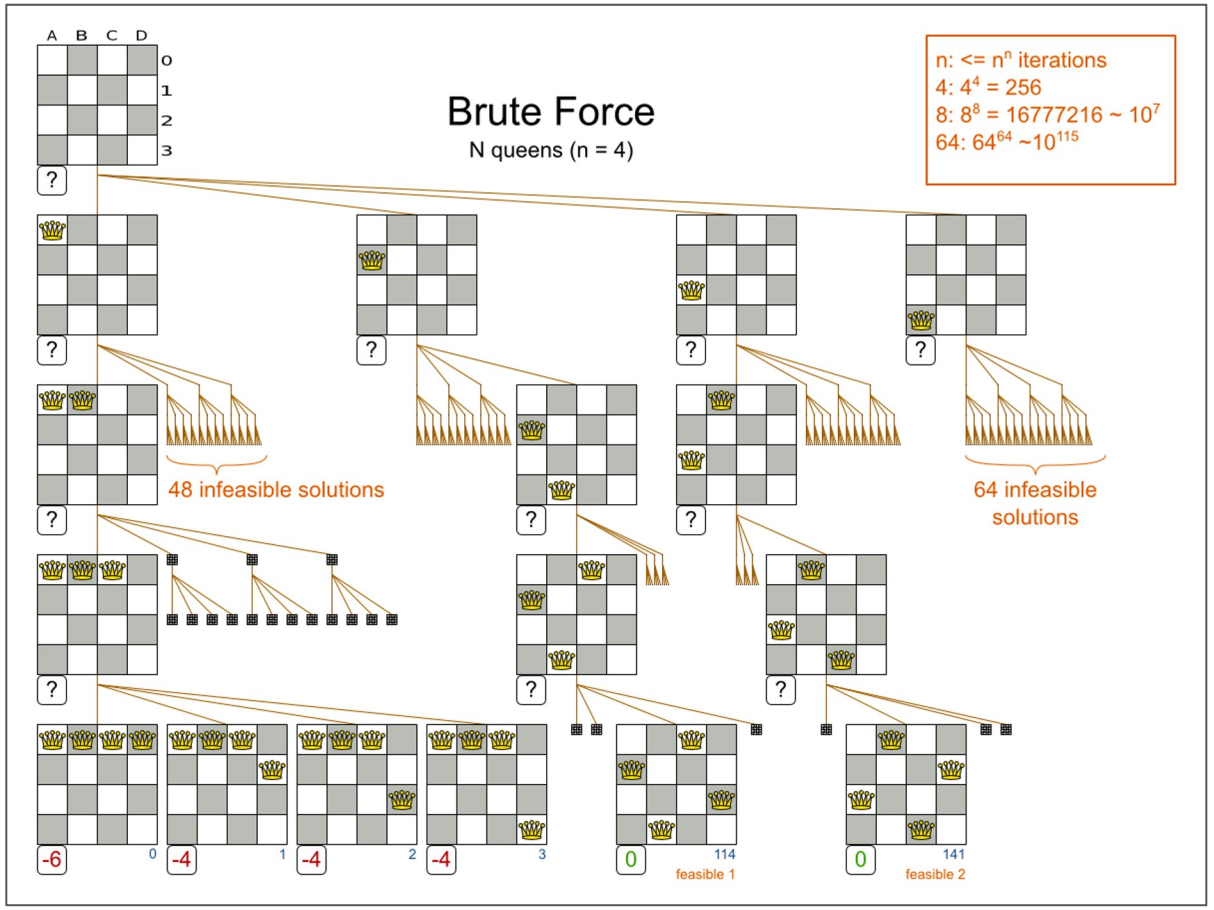


Image credit - https://docs.optaplanner.org/

The Brute Force algorithm creates and evaluates every possible solution.

Notice that it creates a search tree that explodes exponentially as the problem size increases, so it hits a scalability wall.

Exhaustive Search variants suffer from 2 big scalability issues:

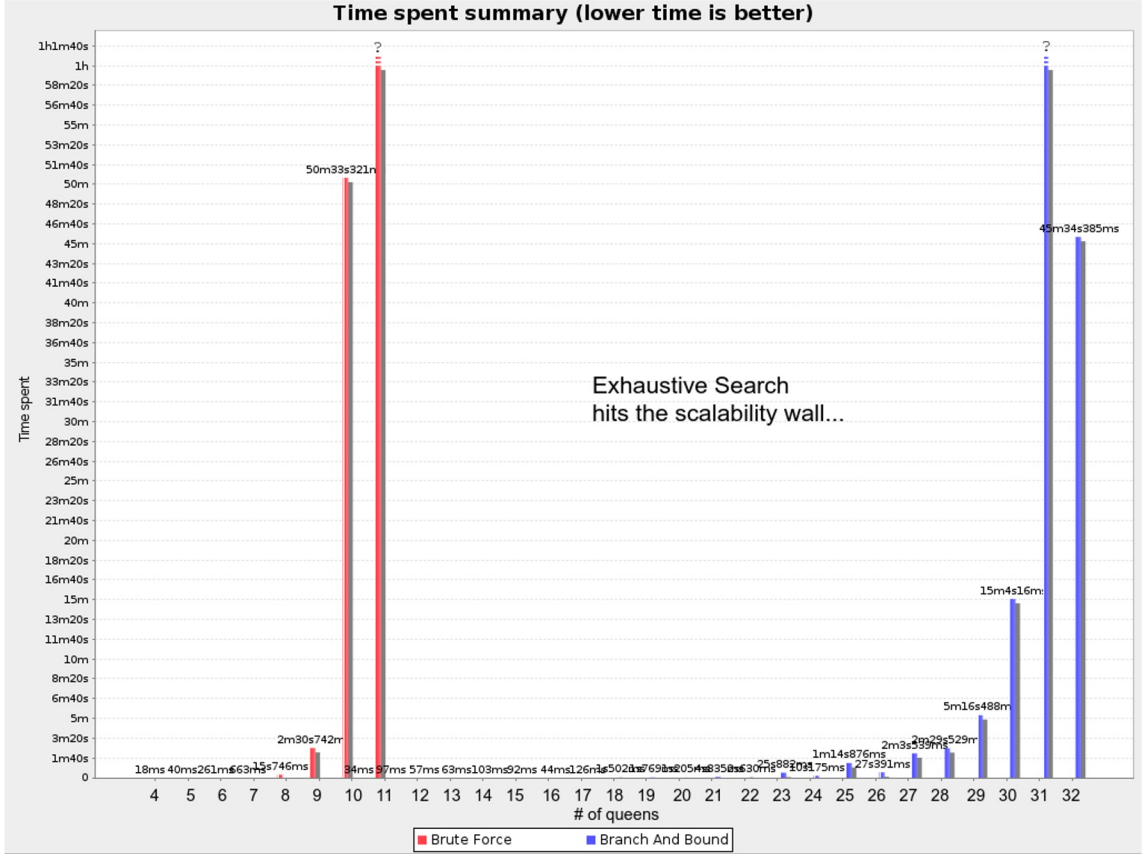
- They scale terribly memory wise.
- They scale horribly performance wise.

The advantages of the brute-force search are:

- Will find the correct solution from all possibilities
- Useful for small and simple problems
- Does not require domain knowledge

Brute Force or Exhaustive Search

Scalability of Exhaustive Search



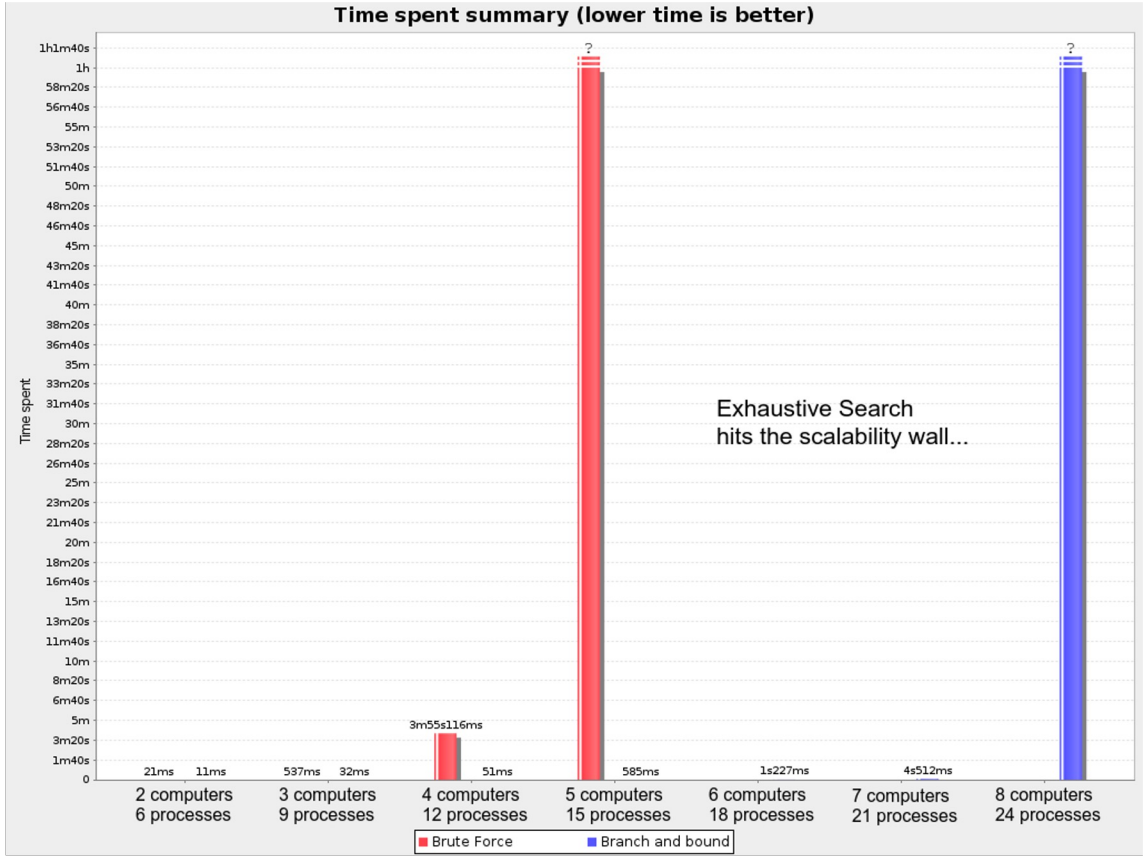


Image credit - https://docs.optaplanner.org/

Brute Force or Exhaustive Search

Combinatorial Explosion

In mathematics, a combinatorial explosion is the rapid growth of the complexity of a problem due to how the combinatorics of the problem is affected by the input, constraints, and bounds of the problem. Combinatorial explosion is sometimes used to justify the intractability of certain problems.

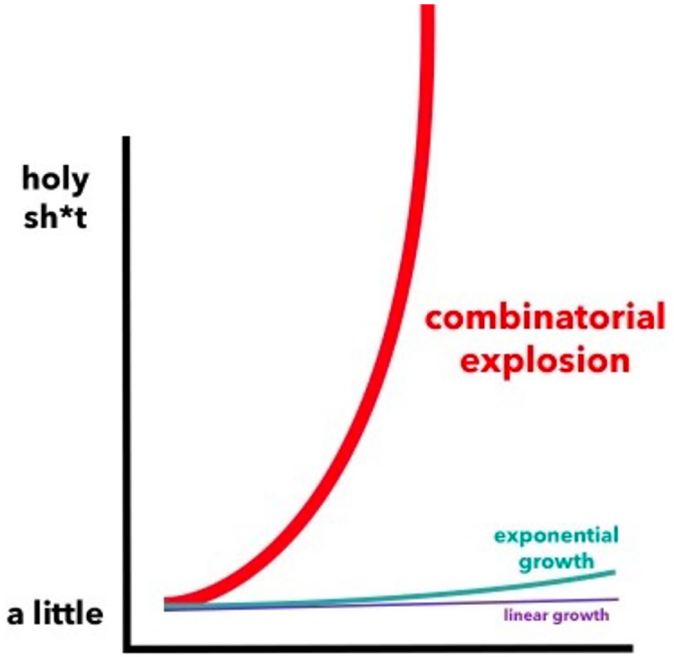


Image credit - https://www.freecodecamp.org/

In the game of chess the number of choices at each level increases by the branching factor, which may typically multiply the options by 20 or more at each move.

Although in theory it should be possible to analyze the game of chess from start to finish, the number of states to be examined is so enormous that it is completely impractical, not only at present but for any conceivable computer in the future.

To appreciate this, consider an example: if one million game states can be examined each second and the branching factor is 10, then to analyze 6 moves ahead takes 1 second, to analyze 12 moves takes 11 days, and to cover 18 moves takes nearly 32 000 years.

One of the main thrusts of artificial intelligence work has been to find ways, such as **heuristic search**, to circumvent the combinatorial explosion.

AlphaZero and AlphaGo

AlphaZero was developed by the artificial intelligence and research company DeepMind, which was acquired by Google. It is a computer program that reached a virtually unthinkable level of play using only reinforcement learning and self-play in order to train its neural networks. In other words, it was only given the rules of the game and then played against itself many millions of times (**44 million games in the first nine hours**, according to DeepMind).

Google DeepMind

Resource: https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go

Image credit - DeepMind Technologies

Go - There are an astonishing 10 to the power of 170 possible board configurations - more than the number of atoms in the known universe.
https://www.deepmind.com/research/highlighted-research/alphago

Travelling Salesman Problem

- One of the most famous Computer Science problems in the field of combinatorial optimization
- Given n cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- It is used as a benchmark for many optimization methods
- Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%

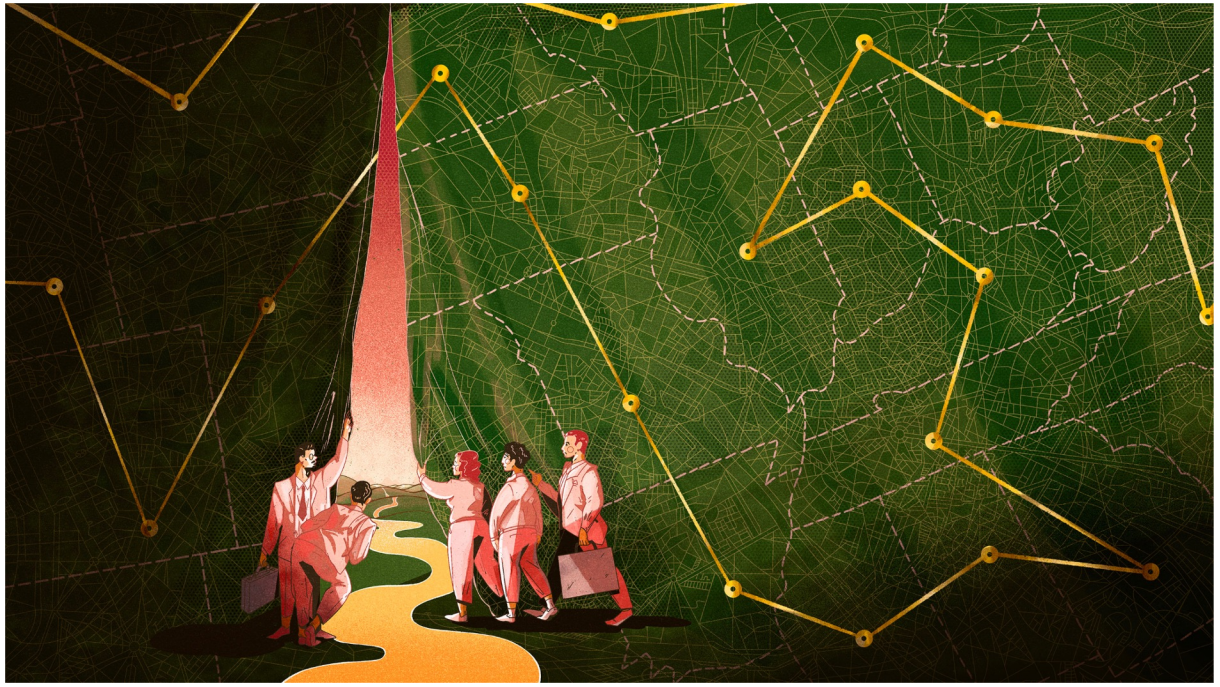


Image credit - Islenia Mil for Quanta Magazine

Computer Scientists Break Traveling Salesperson Record
https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/

Travelling Salesman Problem

Starting at A, find the shortest path through all the cities, visiting each city exactly once and returning to A.

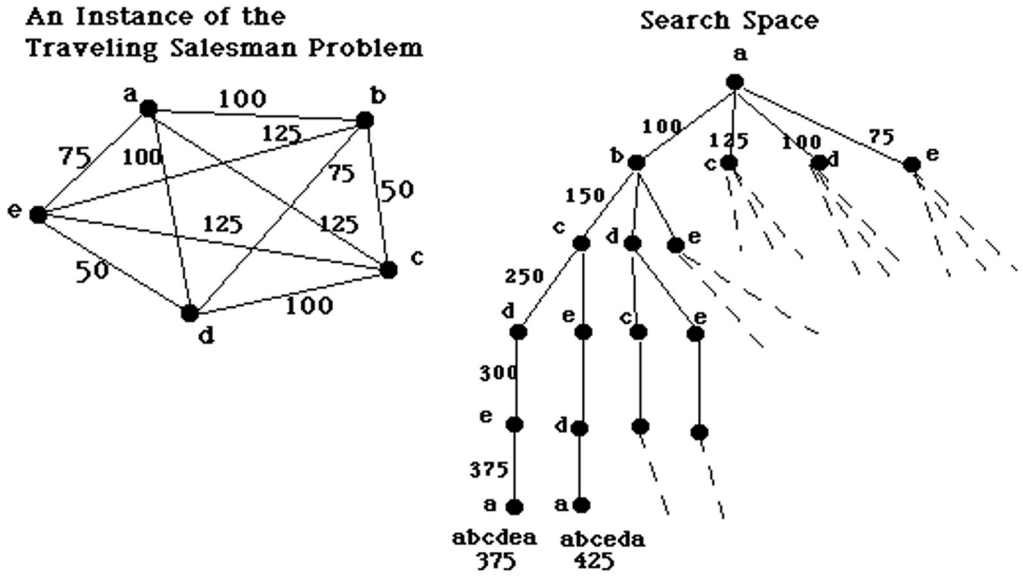


Image credit - http://www.cs.trincoll.edu/

- Traveling salesman problem: why visit each city only once? https://math.stackexchange.com/questions/1269983/traveling-salesman-problem-why-visit-each-city-only-once

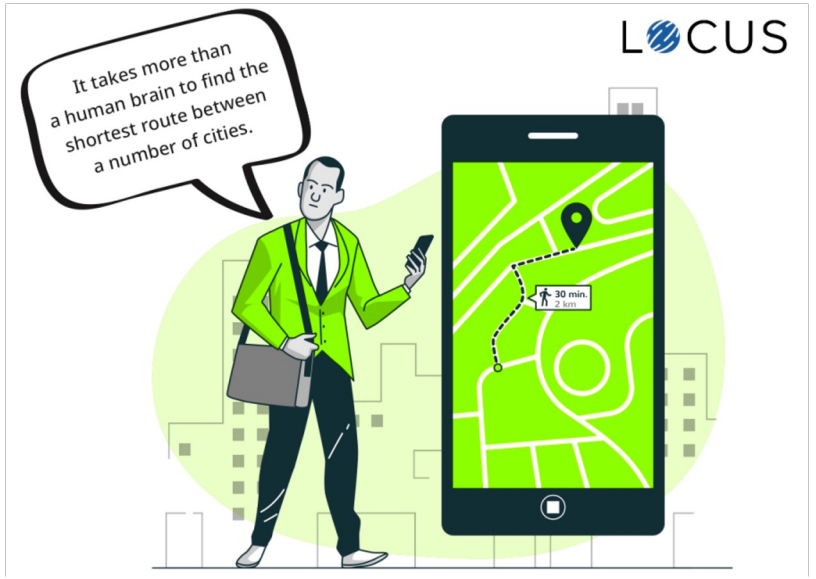


Image credit - https://blog.locus.sh/

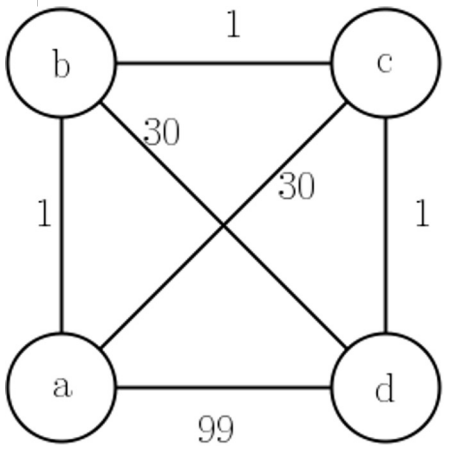
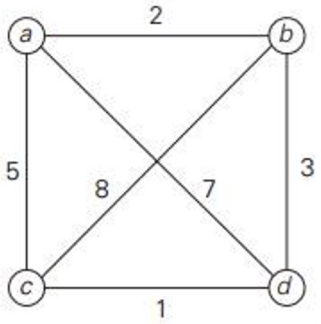


Image credit - https://cs.stackexchange.com/

Nearest Neighbor (Greedy Algorithm) does not guarantee an optimal solution.

Brute Force or Exhaustive Search

Travelling Salesman Problem



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search.

Image credit - https://www.brainkart.com/

The formula to calculate the number of distinct tour paths for **n** cities is **(n-1)!**

A four city tour has six possible routes (3 x 2 x 1)/2=3, whilst a five city tour has 12 possible distinct tours (4 x 3 x 2 x 1)/2=12.

A problem whose possibilities increase at such a rate rapidly becomes complex. As it would require 60 distinct drawings to represent a six city problem, humans are understandably turning toward the superior calculating speed of the computer to help with the problem.

The optimum tour can be found by calculating the total length of each possible route around the cities and choosing the shortest of those. But even the computer begins to struggle at a relatively low number to consider all possible solutions. Finding the optimal route for a 30 city tour would require the calculation of distance of 4.42×10^{30} different possible tours.

Brute Force or Exhaustive Search

Assignment Problem

Linear Assignment problems are fundamental combinatorial optimization problems. In most general form, the problem instance has a number of agents and a number of tasks.

Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment.

It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized.

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs
   (take minimum entry from each row).

2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers
   (take minimum entry from each column).

Suppose that a taxi firm has three taxis (the agents) available, and three customers (the tasks) wishing to be picked up as soon as possible.

For each taxi the "cost" of picking up a particular customer will depend on the time taken for the taxi to reach the pickup point. This is a balanced assignment problem. Its solution is whichever combination of taxis and customers results in the least total cost.

Now, suppose that there are four taxis available, but still only three customers. This is an unbalanced assignment problem.

One way to solve it is to invent a fourth dummy task, perhaps called "sitting still doing nothing", with a cost of 0 for the taxi assigned to it. This reduces the problem to a balanced assignment problem, which can then be solved in the usual way and still give the best solution to the problem.

Brute Force or Exhaustive Search

Assignment Problem

Let's take below example and try to calculate promising cost when Job 2 is assigned to worker A.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker C as it is only Job left. Total cost becomes 2 + 3 + 5 + 4 = 14.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Search space diagram showing optimal solution path in green.

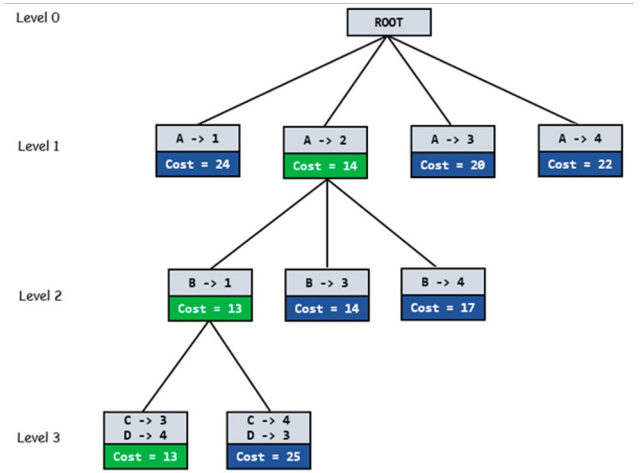


Image credit - https://www.geeksforgeeks.org/

Brute Force or Exhaustive Search

Assignment Problem

A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search.

<1, 2, 3, 4> indicates the assignment of

- Person 1 to Job 1,
- Person 2 to Job 2,
- Person 3 to Job 3,
- Person 4 to Job 4.

Similarly,

<1, 2, 4, 3> indicates the assignment of

- Person 1 to Job 1,
- Person 2 to Job 2,
- Person 3 to Job 4,
- Person 4 to Job 3.

Since the number of permutations to be considered for the general case of the assignment problem is ***n***!, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the ***Hungarian method*** after the Hungarian mathematicians Dénes Kőnig and Jenő Egerváry,´ whose work underlies the method

Hungarian Algorithm Introduction & Python Implementation
https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementation-93e7c0890e15

Brute Force or Exhaustive Search

0-1 Knapsack Problem

- The knapsack problem is a problem in combinatorial optimization. The knapsack problem has been studied for more than a century, with early works dating as far back as 1897.
- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- For n items, exhaustive search leads to a (2^n) algorithm, no matter how efficiently individual subsets are generated.

Image credit - https://www.askpython.com/

Problem details

Suppose we have a knapsack which can hold `int w = 10` weight units. We have a total of `int n = 4` items to choose from, whose values are represented by an array `int[] val = {10, 40, 30, 50}` and weights represented by an array `int[] wt = {5, 4, 6, 3}`.

Since this is the 0–1 knapsack problem, we can either include an item in our knapsack or exclude it, but not include a *fraction* of it, or include it *multiple* times.

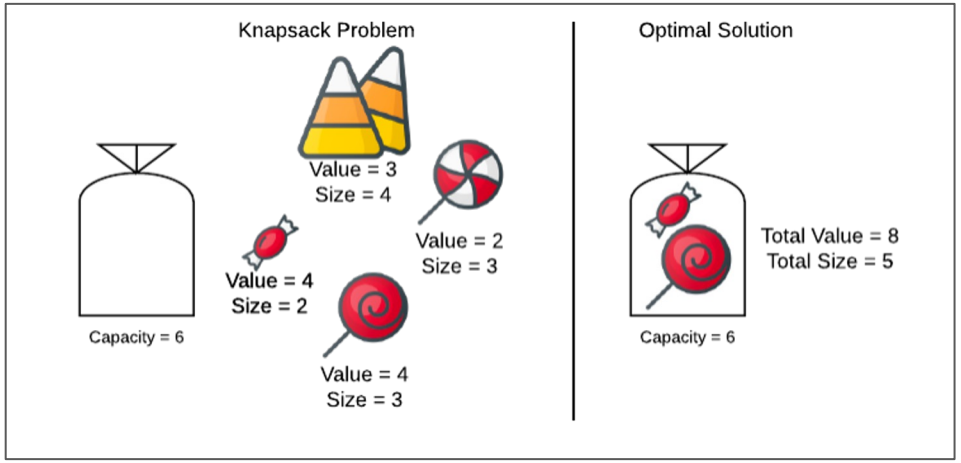
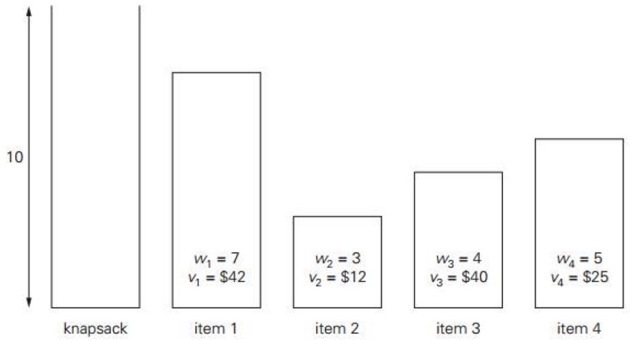


Image credit - https://hideoushumpbackfreak.com/

Brute Force or Exhaustive Search

0-1 Knapsack Problem



(a)

Subset	Total weight	Total value
∅	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

Image credit - https://www.brainkart.com/

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

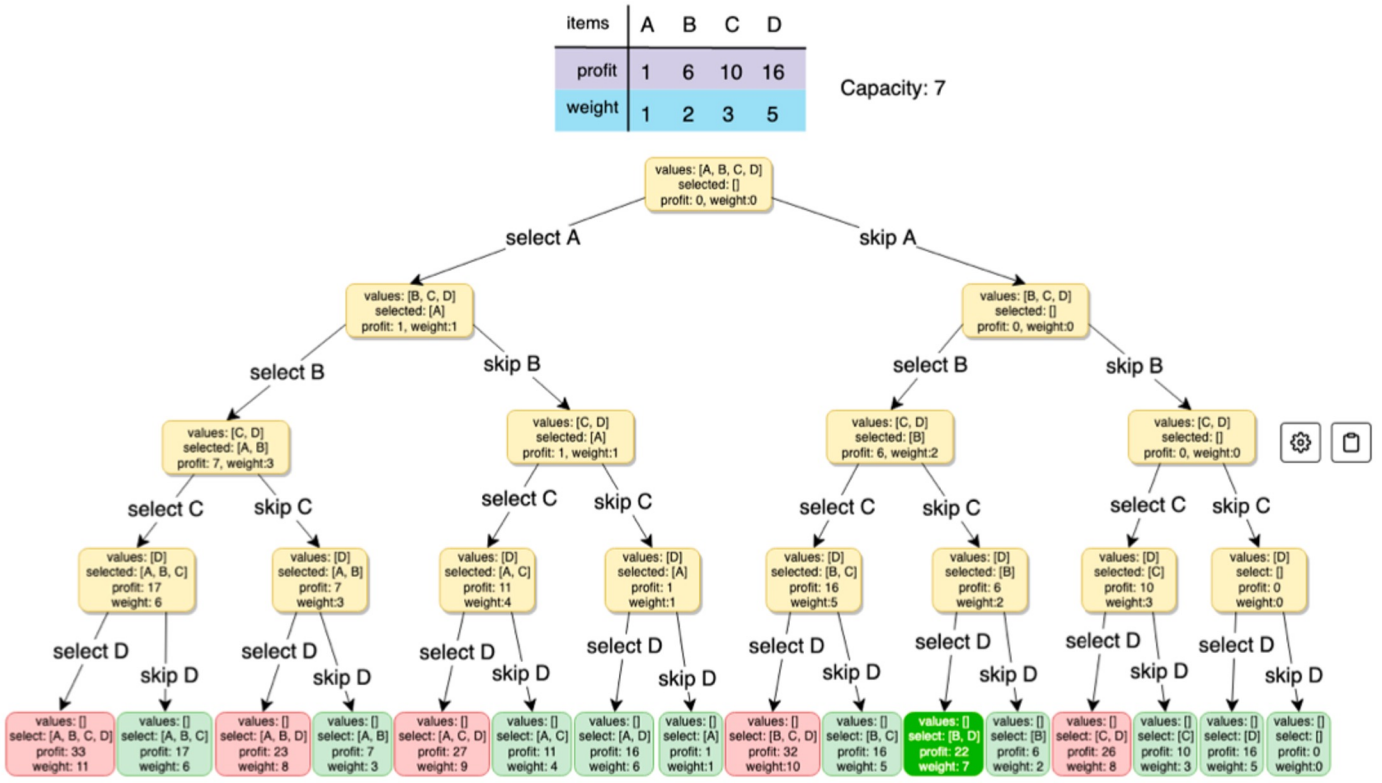


Image credit - https://astikanand.github.io/
Article - https://astikanand.github.io/techblogs/dynamic-programming-patterns/0-1-knapsack-pattern