Algorithms and Data Structures

Bonus Lecture

Teaching Assistant: Steven Abreu

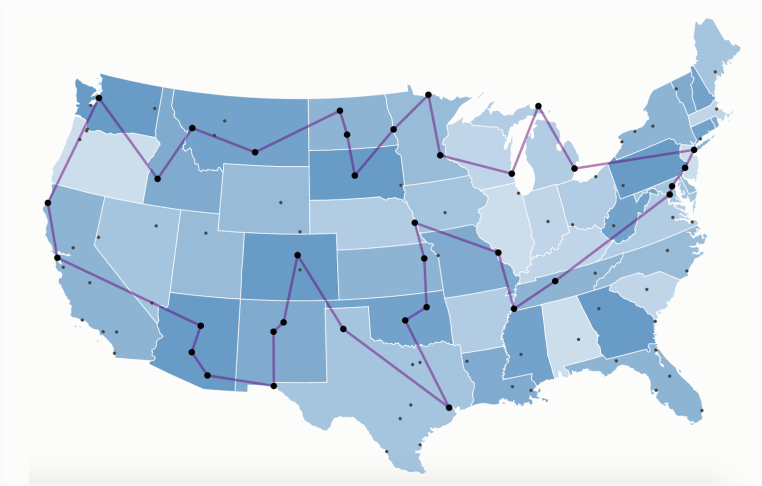
Jacobs University Bremen

Table of contents

Traveling Salesman Problem

Problem Description

Given some cities, find the shortest route that visits each city once.



Relevance of TSP

TSP is the most famous **NP-hard** problem. This means that there exists no polynomial time algorithm that solves this problem (not yet).

There are $\frac{1}{2}(n-1)!$ possible routes for n cities:

Number of cities	Number of different routes
1	1
5	12
10	181.440
20	$6.08 * 10^{16}$
50	$3.04 * 10^{62}$

NP Completeness

The Traveling Salesman Problem is an NP-complete problem. This means that it is "representative" for the class of NP problems - the set of all problems which cannot be solved in polynomial time.

This means that every NP-hard problem can be converted to an instance of TSP in polynomial time.

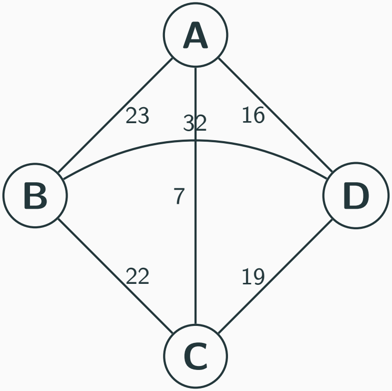
If we manage to solve TSP in polynomial time, we would be able to solve any NP-hard problem in polynomial time - those are the most interesting problems.

TSP Applications

- Logistics and Planning (quite obviously)
- Circuit board manufacturing: determine in which order to drill holes.
- Astronomy: minimize time spent moving telescope between observations.
- Many more (DNA sequencing, Routing, X-Ray, ..)

Graph Representation

The TSP can be represented with a graph, like so:



Problem Formulation - Graph version

Consider the (complete, undirected) graph $G = (V, E)$:

- Vertices V (numbered $1, .., n$).
- Edges $E = \{(x, y) \mid x, y \in V\}$.
- Distance function $d : E \to \mathbb{N}_+$.

We denote our solution like this:

- Let $\mathbb{H}$ be the set of all Hamiltonian cycles (cycles that visit each node exactly once).
- We want to find $h_{optimal} \in \mathbb{H}$, such that the sum of distances in that tour is minimized.

Solution - Brute Force Search

We can solve this problem easily using brute force search, checking all possible routes:

1. Consider city 1 to be the starting city (route is cyclic).
2. Generate list of all (n-1)! permutations of the cities.
3. Calculate cost of each permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Time complexity: $O(n!)$

$\rightarrow$ Optimal, but inefficient.

Solution - Dynamic Programming (1)

We will now look at the most famous exact solution of the TSP, the **Held-Karp algorithm.**

As in all dynamic programming algorithms, we solve the problem by solving the smallest subproblems and then using those solutions to solve increasingly large problems (until we reach the original problem).

This works similar to the dynamic programming algorithm for Fibonacci numbers.

Solution - Dynamic Programming (2)

- Let us number the cities with $1, .., N$.
- The distance between city i and city j are denoted by d_{ij}.
- Consider subsets $S \subseteq \{2, .., N\}$ of cities.
- For $c \in S$, let $D(S, c)$ be the minimum distance from city 1, visiting all cities in S, finishing in city c.

Solution - Dynamic Programming (3)

Steps towards the solution:

- Phase 1: $S = \{\}$ and $D(S, c) = d_\{1, c\}$ for all cities c.
- Phase 2: $S = \{x\}$ (iterate through cities) and
  $D(S, c) = d_\{c, x\} + D(\{\}, c)$ for all cities c.
- Phase k: $S = \{...\}$, with $|S| = k$ and
  $D(S, c) = \min_{x \in S} (D(S \setminus \{x\}, x) + d_{x,c})$ for all cities c.

From this, we can see that we need to iteratively compute the distances $D(S, c)$ for increasingly large S.

Problem Formulation - Integer Programming version (1)

Let us denote all n cities by indices $i, .., n$, the distance between city i and j by c_{ij}. We introduce the decision variables y_{ij} such that

$$y_{ij} = \begin{cases} 1 & \text{if city } j \text{ is visited immediately after city } i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We want to minimize the objective function

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} y_{ij} \tag{2}$$

Problem Formulation - Integer Programming version (2)

We add the following constraints to our problem formulation:

Go-to constraint: after visiting city i, we can only visit one city next

$$\forall i \in \{1, .., n\} : \sum_{j=1}^{n} y_{ij} = 1 \tag{3}$$

Come-from constraint: when visiting a city, we can only come from one city

$$\forall j \in \{1, .., n\} : \sum_{i=1}^{n} y_{ij} = 1 \tag{4}$$

Subtour elimination: Do not allow disjointed subtours

$$\forall S \subset V,\ S \neq \emptyset : \sum_{i=1}^{n} \sum_{j=1}^{n} y_{ij} \leq |S| - 1 \tag{5}$$

Solution - Simplex method

The integer programming formulation of the TSP is often solved using the Simplex algorithm. This can be done using software packages like Pyomo.

We will not cover that today. If you're interested, consider taking the course 'Operations Research' from the joint IMS/IEM CORE module offered by Prof. Marcel Oliver.

Heuristic Solutions

Since solving the TSP takes so much time, we are also interested in approximating the solutions, if that saves us a significant amount of time.
Such approximative algorithms are called **heuristics**.
We will introduce the following two heuristic solutions for the TSP:

- Nearest Neighbors algorithm
- A better Nearest Neighbors algorithm
- Greedy algorithm

Heuristic Solution - Nearest Neighbors

We can also solve this problem using the **Nearest Neighbors** algorithm, a greedy algorithm.

The idea: after visiting each city, choose the closest city that you haven't visited yet.

This does not always return the optimal solution.

This solution is within 25% to the exact solution. $\rightarrow$ Efficient, but

non-optimal. This is an **approximate algorithm**, also called a **heuristic**.

Heuristic Solution - Better Nearest Neighbors

We can improve our Nearest Neighbor algorithm to the **Repeated Nearest Neighbor** algorithm by running the Nearest Neighbor algorithm with every possible starting city.

```
min_path, min_cost = None, inf
for i in cities:
        path, cost = nearest_neighbor(start=i)
        if cost < min_cost:
                min_path, min_cost = path, cost
```

Better, but still non-optimal.

Heuristic Solution - Greedy Algorithm

- Sort all edges by their distances.
- Add the cheapest edge (if no cycle with less than N edges and if no node has degree bigger than 2).
- If we have less than N edges, repeat step 2.

This solution is within 15-20% to the exact solution.

Perfect TicTacToe Player using the Minimax algorithm

Representing games with a tree (1)

Many games can be represented as a tree. We consider a game that involves no luck (i.e. no dice-rolling) and no hidden information (i.e. no bluffing). We represent the course of the game with a tree:

- Every node represents a turn.
    - Root node: initial state
    - Nodes of even depth: turn of player 1
    - Nodes of odd depth: turn of player 2
    - Leafs: terminal states (game is over)
- For every node n, its children are the possible moves in that turn.
- Every branch represents a possible course of the game.

Representing games with a tree (2)

For every leaf l, let $x = score(l) \in \mathbb{Z}^{\infty}$ represent the outcome:

- $x = \infty$: player 1 wins
- $x > 0$: player 1 is ahead
- $x = 0$: draw
- $x < 0$: player 2 is ahead
- $x = -\infty$: player 2 wins

Thus, player 1 wants to maximize x, while player 2 wants to minimize x.

The minimax algorithm (1)

The minimax algorithm builds the entire tree by exploring all possible games.

Let us assume to have the game tree $\mathit{game} : \mathit{Tree}[\mathit{State}]$, with State being the game state. We define the functions

- $\mathit{isTerminal} : \mathit{State} \to \mathit{Bool}$ to check if the state is terminal
- $\mathit{result} : \mathit{State} \to \mathbb{Z}^{\infty}$ to get the result of the game (previous slide).

The minimax algorithm (2)

```
def minimax(current, depth):
  """ current: Tree[State]
      depth: number in [0,1,2,...]
      Returns the score of the optimal outcome.
  """
  state = current.data
  if isTerminal(state):
    return result(state)
  childResults = map(current.children, lambda n: minimax(n,
      depth + 1))
  if even(depth):
    return max(childResults)
  else:
    return min(childResults)
```

The minimax algorithm (3)

Let $\mathit{res} = \mathit{minimax}(\mathit{game}, 0)$ be the result of the minimax algorithm:

- $\mathit{res} = \infty$: player 1 has a perfect strategy to win every game.
- $\mathit{res} = 0$: both players have perfect strategies to draw every game.
- $\mathit{res} = -\infty$: player 2 has a perfect strategy to win every time.

Applications of the minimax algorithm

In the Tic Tac Toe game, we can build the entire tree and compute the perfect strategy in reasonable time. There are $9! = 362.880$ possible games to be played (why?).

In most other games (e.g. chess), the game tree would be far too big to be built. We can still use the minimax algorithm, but instead of computing the entire tree, we define a cut-off depth, after which we estimate the result using a heuristic function.

Solving Sudoku using the Backtracking algorithm

Backtracking overview

If we have a problem involving multiple successive choices (tree-like structure), a backtracking algorithm works as follows:

1. Choose one of the options and proceed.
2. If we reach a dead-end, revert back to the previous state and choose a different option.

The algorithm makes sense if we have little (or no) information about what choice to make and if we can detect quickly if we make a wrong choice.

Backtracking pseudocode

We define the following functions:

- *abort* : *state* $\rightarrow$ *bool*: returns true if the current state is a dead end.
- *solution* : *state* $\rightarrow$ *bool*: returns true if the current state is a solution.
- *choices* : *state* $\rightarrow$ *state*[]: returns the list of possible next moves.

```
def search (state: List[A]) -> List[A]?:
        if abort(state) { return None }
        if solution(state) { return state }
        for choice in choices(state):
                x = search(state + [choice])
                if x != None { return x }
        return None
```

8 queens problem

One example problem for backtracking is the 8-queens problem:

Place 8 queens on a chess board with none of them threatening another.

Since there has to be one queen in each row, we can represent the solution with a list $[c_1, .., c_8]$, such that c_i is the column coordinate of the queen in row i.

The possible choices are defined like so:

$$choices(state) = \begin{cases} [1,2,3,4,5,6,7,8] & length(state) < 8 \\ [] & length(state) = 8 \end{cases} \tag{6}$$

The function $abort(state)$ returns true if two queens threaten each other.

Solving a Sudoku board

In the Sudoku game, we have 81 variables $x_{ij} \in \{1, .., 9\}$ for $i, j = 1, .., 9$.

We have 27 constraints in the game: each of the nine rows, columns and 3x3 squares need to contain all numbers from 1 to 9. If any of these constraints are violated, then the *abort* function returns true.

The *solution* function returns true if no contraints are violated and the board has no empty fields.

We go through the board field by field (skipping the ones that are given), thus the *choices* function returns a number from 1-9 for the next field.