Traveling Salesman Problem

A shallow dive exploring solutions from traditional approaches to artificial intelligence

Presentation by: Devid Duma

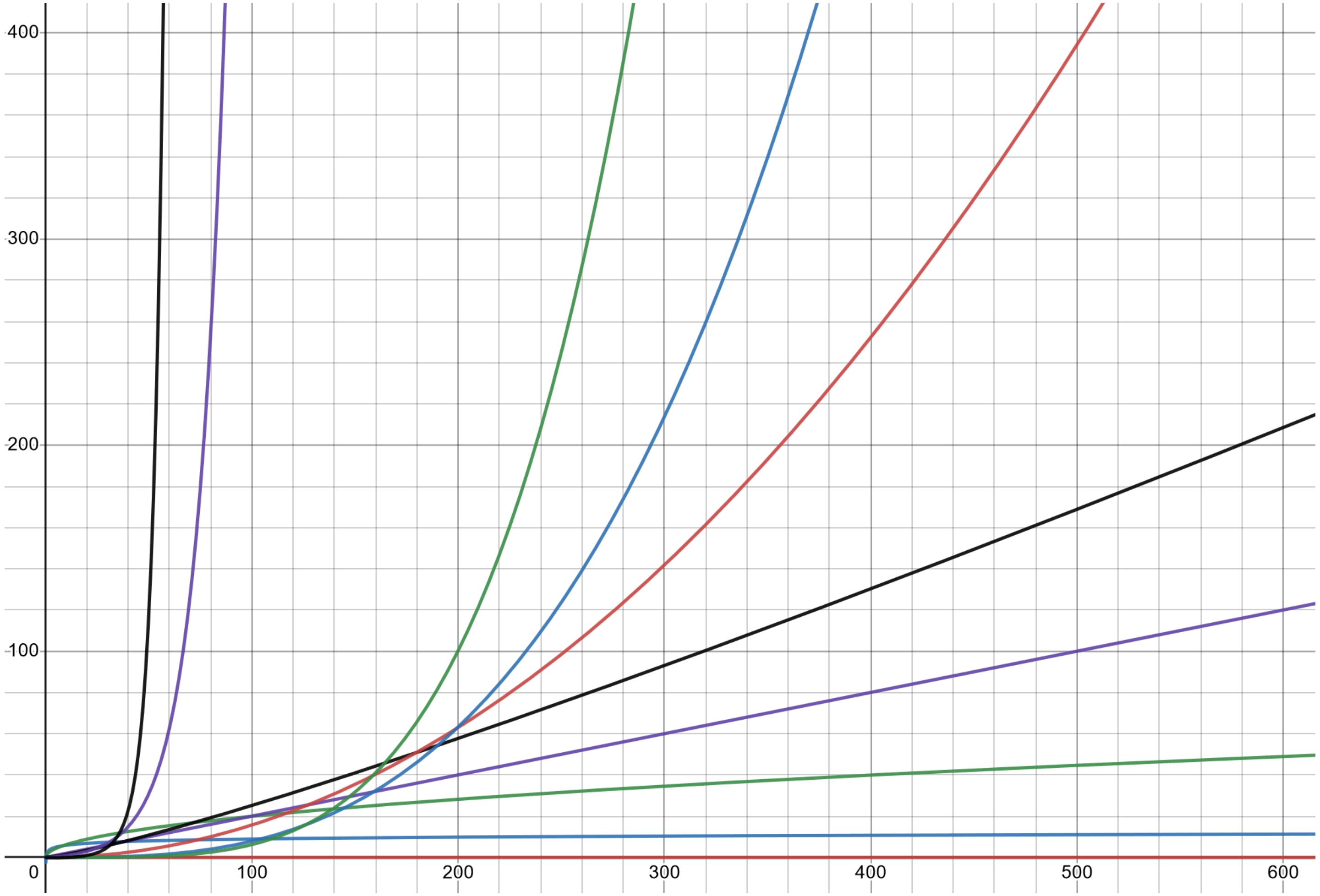
Formulation of TSP intuitive

- The traveling salesman problem (TSP) asks the question:
- "Given a list of cities and the distances between each pair of cities, what is the **shortest** possible route (**permutation of cities**) that visits each city and returns to the origin city?“

- TSP is a **combinatorial** problem.
  - Naïve brute force yields $\boldsymbol{\Theta(n!)}$ time complexity.

Graphs of different time complexities

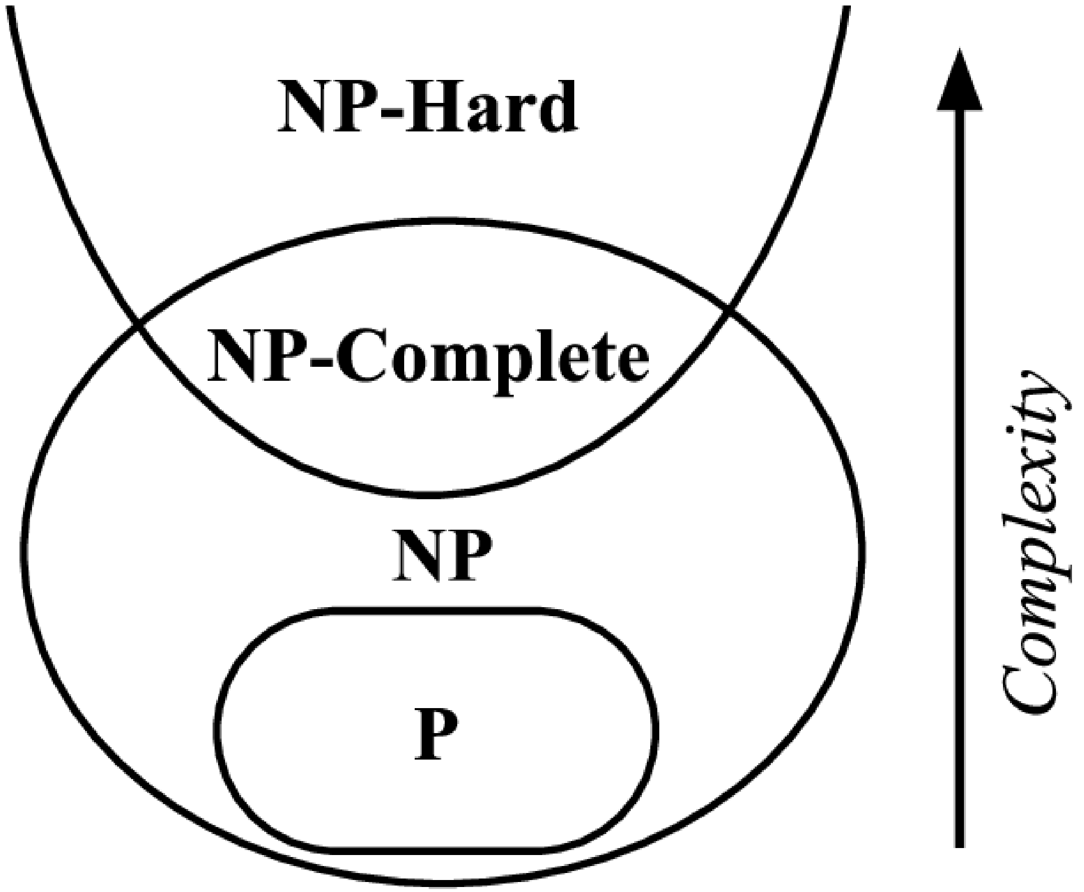
- Link to Desmos Graph.
- $\Theta(1)$
- $\Theta(\log n)$
- $\Theta(\sqrt{n})$
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(n^3)$
- $\Theta(n^4)$ polynomial time
- $\Theta(2^n)$ exponential time
- $\Theta(n!)$ TSP



Asymmetric TSP

- The key difference between symmetric TSP and asymmetric-TSP is that the distance function may not be symmetric.
- That is, for two locations u and v, it is possible that $d(u,v) \neq d(v,u)$.

P, NP, NP-complete & NP-hard



P vs NP
intuitive definitions

- Source: Stackoverflow

- **Decision problem**: A problem with a yes or no answer.
- **P** is a complexity class that represents the set of all **decision problems** that can be solved in **polynomial time**.
- **NP** is a complexity class that represents the set of all **decision problems** for which the instances where the **answer is "yes"** have **proofs** that can be **verified** in polynomial time.
  - if someone gives us an instance of the problem and a certificate (sometimes called a **witness**) to the answer being yes, we can **check** that it is correct in **polynomial time**.

NP-complete vs NP-hard intuitive definitions

- Source: Stackoverflow

- **NP-Complete** problems are **part of NP**, as well as represent the set of all problems X in NP for which it is possible to **reduce** any other NP problem Y to X in **polynomial time**.
  - Example: The *satisfiability problem*, *Hamiltonian-Cycle problem*
- A problem X is **NP-hard**, if there is an NP-complete problem Y, such that Y is **reducible** to X in **polynomial time**.
  - NP-hard problems **do not have to be in NP**, and they **do not have to be decision problems**.

Formulation of TSP formal

- Hamiltonian Cycle Problem: **NP-complete**
  - Given as input an unweighted undirected graph $G = (V, E)$, is there a Hamiltonian cycle in G?
- Decision-TSP (TSP Search): **NP-complete**
  - Given as input a weighted undirected graph $G = (V, E, \omega)$ and at **bound $\boldsymbol{C}$**, is there a Hamiltonian cycle in G whose weight is at most C?
- Optimization-TSP: **NP-hard**
  - Given as input a weighted undirected graph $G = (V, E, \omega)$, find the Hamiltonian cycle in G whose weight is **minimal**.

D-TSP is NP-complete, TSP-OPT is NP-hard

- Hamiltonian Cycle Problem, which is NP-complete, can be reduced in polynomial time into Decision-TSP.
  - Proof shown at math.stackexchange.com
  - We can take any instance $G = (V, E)$ for the Hamiltonian cycle problem and convert it into an instance $G' = (V, E' = V \times V, \omega), C = 0$ of Decision-TSP.

- Decision-TSP is NP-complete:
  - a **decision problem**
  - **can be reduced further** to TSP-OPT
  - given a tour by a witness, we can check that the tour contains each vertex once, sum the total cost of the edges and finally check if the cost is minimum. All this can be completed in *polynomial time*, thus D-TSP **belongs to NP**.

Traditional Approaches

- **Exhaustive Algorithms**: Will always find the best possible solution by evaluating every possible path.
  - Random Walk, Depth First Search, Branch and Bound
- **Heuristics**: Attempt to find a good approximation of the optimal path within a more *reasonable* amount of time.
  - *Constructive:* Nearest Neighbor, Cheapest Insertion, Furthest Insertion, Nearest Insertion, Convex Hull Insertion, Simulated Annealing
  - *Improvement:* 2-Opt Reciprocal Exchange, 2-Opt Inversion

- *Approximating bounds*: using Minimum Spanning Tree
- Github Visualisation at tspvis.com

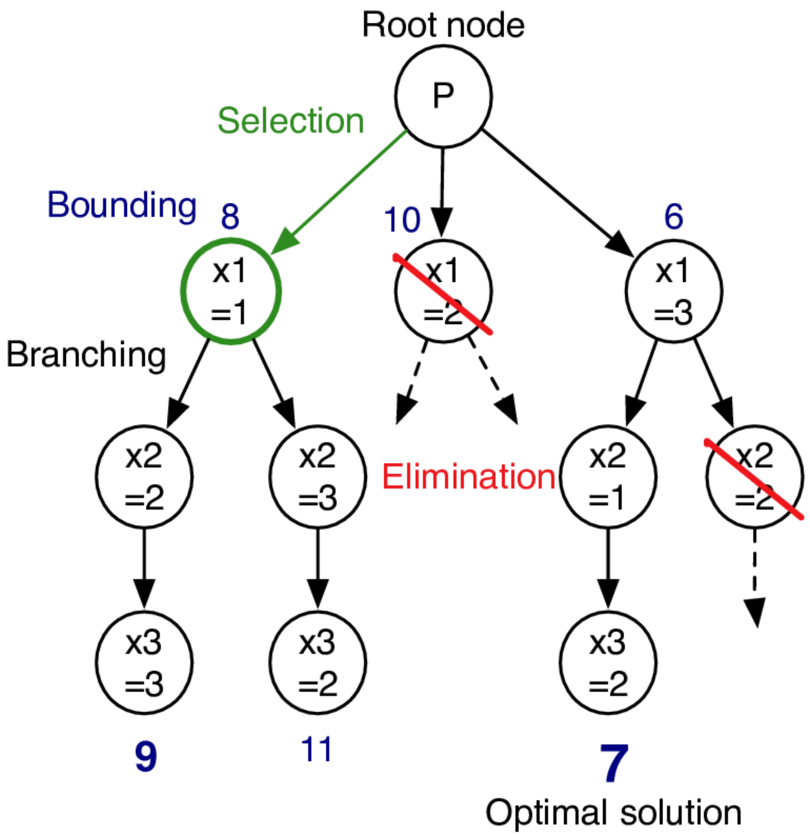
Exhaustive Algorithms

- **Random Walk**
  - Construct a random cycle according to a pseudo-random algorithm. Repeat until you find an optimal path.
  - Could be enhanced with **dynamic programming**.
- **Depth First Search**
  - The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

- Time complexity: worst case $\Theta(n!)$

Exhaustive Algorithms

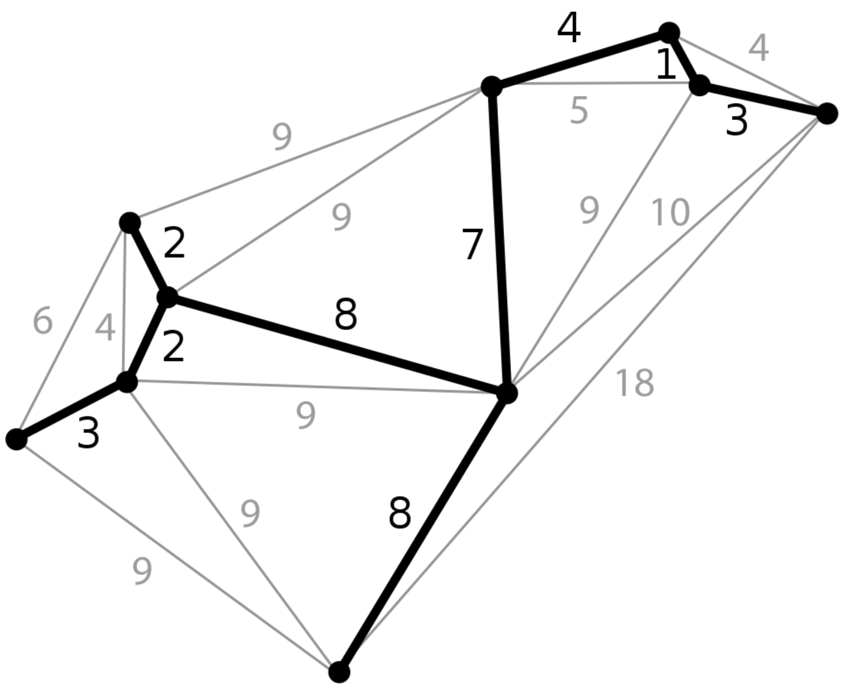
- **Branch and Bound**
  - The set of candidate solutions is thought of as forming a rooted tree with the full set at the root.
  - Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution.
  - It is discarded if it cannot produce a better solution than the best one found so far by the algorithm.



- [BnB Demo]()
- Time complexity: worst case $\Theta(n!)$

Approximating Initial Bounds

- **Minimum Spanning Tree** is a subset of the edges of a connected, edge-weighted undirected graph that:
  - connects all the vertices together
  - without any cycles
  - with the **minimum** possible total edge weight



- Kruskal's Algorithm / Prim's Algorithm
  - $\Theta(n \log n)$

Approximating Initial Bounds

- Initial **Upper Bound** approximation: Demo
  - $2 * MST$ (Minimum Spanning Tree)

- Initial **Lower Bound** approximation: Demo
  - Randomly choose a vertex v from graph G and remove it.
  - The remaining graph is a Residual Tree. Calculate $RMST$ (**Residual Minimum Spanning Tree**) of Residual Tree.
  - Add vertex v back in the graph, along with the two shortest distinct arcs of v (vi, vj) connecting it to G. The sum $RMST + vi + vj$ is a **candidate** initial lower bound.
  - Repeat **for each vertex** v in G and return the maximum candidate.

Heuristics

- **Nearest Neighbor**
  - TSP starts at a random city and repeatedly visits the nearest city until all have been visited. Quick approximation, although not optimal. NN Demo
  - Time complexity: $\Theta(n^3)$

- **Nearest Insertion**
  - Start with a sub-graph consisting of node i only. Find node r such that c_{ir} is minimal and form sub-tour $i - r - i$.
  - *(Selection step)* Given a sub-tour, find node r **not** in the sub-tour **closest** to **any** node j in the sub-tour (i.e. with minimal c_{rj})
  - *(Insertion step)* **Find the arc** $(\boldsymbol{i}, \boldsymbol{j})$ in the sub-tour which minimizes $c_{ir} + c_{rj} - c_{ij}$ and insert r between i and j. Repeat step 2.
  - Time complexity: $\Theta(n^2)$

Heuristics

- **Cheapest Insertion**
  - Start with a sub-graph consisting of node i only. Find node r such that c_{ir} is minimal and form sub-tour $i - r - i$.
  - *(Insertion step)* Find r **not** in the sub-tour and the **arc** $(\boldsymbol{i}, \boldsymbol{j})$ in the sub-tour which minimizes $c_{ir} + c_{rj} - c_{ij}$. Insert r between i and j. Repeat.
  - Time complexity: $\Theta(n^2)$

- **Farthest Insertion**
  - Start with a sub-graph consisting of node i only. Find node r such that c_{ir} is minimal and form sub-tour $i - r - i$.
  - *(Selection step)* Given a sub-tour, find node r **not** in the sub-tour **farthest** to any node j in the sub-tour (i.e. with maximal c_{rj})
  - *(Insertion step)* Find the arc (i, j) in the sub-tour which minimizes $c_{ir} + c_{rj} - c_{ij}$ and insert r between i and j. Repeat step 2.
  - Time complexity: $\Theta(n^2)$

Heuristics

- **Convex Hull Insertion**: The *convex hull* is the smallest convex polygon completely enclosing a set of points.
  - Form the convex hull of the set of nodes and use this as an initial sub-tour.
  - *(Selection step)* For each node r not in the sub-tour yet, find (i, j) such that $\boldsymbol{c}_{ir} + \boldsymbol{c}_{rj} - \boldsymbol{c}_{ij}$ is minimal. For all (i, j, r) found in step 2, determine (I, R, J) such that $(\boldsymbol{c}_{IR} + \boldsymbol{c}_{RJ})/\,\boldsymbol{c}_{\mathrm{IJ}}$ is minimal.
  - *(Insertion step)* Insert node R in sub-tour between nodes I and J. Repeat step 2.
  - Time complexity: $\Theta(n^2 \log n)$

Heuristics

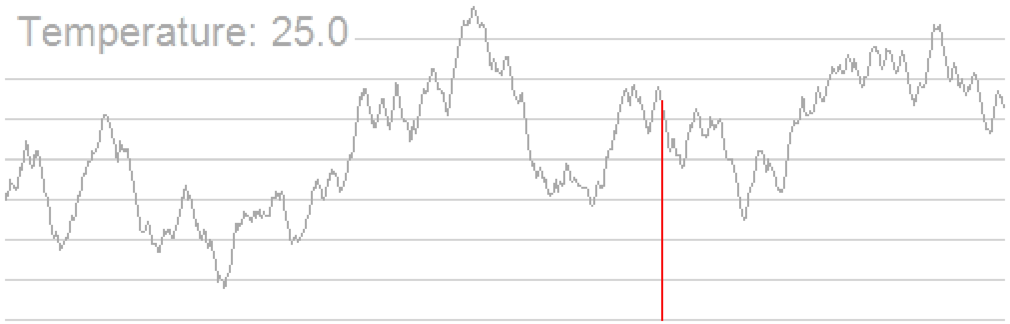
- **2-Opt Inversion**
  - While a better path has not been found:
  - **For each pair of points**:
  - **Reverse** the path between the selected points.
  - If the new path is cheaper (shorter), keep it and continue searching. Remember that we found a better path.
  - If not, revert the path and continue searching.

- **2-Opt Reciprocal Exchange**
  - While a better path has not been found:
  - **For each pair of points**:
  - **Swap** the points in the path. That is, go to point B before point A, continue along the same path, and go to point A where point B was.
  - If the new path is cheaper (shorter), keep it and continue searching. Remember that we found a better path.
  - If not, revert the path and continue searching.

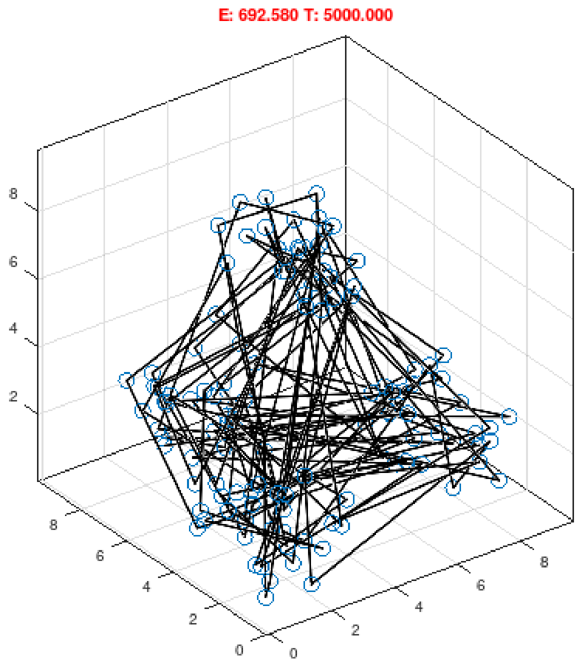
- Time complexity: $\Theta(n^3)$ CUDA acceleration possible according to [this article] [2-opt Intuition]

Heuristics

- **Simulated Annealing**
  - SA is a metaheuristic to approximate global optimization in a large search space for an optimization problem.
  - It is an alternative to Gradient Descent.



When temperature is high, SA explores a lot. As the temperature drops, the algorithms starts exploiting more until it reaches the global optimum when temperature gets to 0.



Using SA to solve TSP in 3D.

Heuristics

- **Simulated Annealing**
  - Start with an initial solution $s = S_0$ and $t = t_0$
  - Define Energy Magnitude formula: $\boldsymbol{P}(\Delta \boldsymbol{c}) = \boldsymbol{e}^{-\frac{\Delta \boldsymbol{c}}{\boldsymbol{t}}}$, Δc is change in cost
  - Setup a temperature reduction function α
    - Linear Reduction Rule: $t = t - \alpha$
    - Geometric Reduction Rule: $t = t * \alpha, 0 < \alpha < 1$
    - Slow Decrease Rule: $t = \frac{t}{1 + \beta t}$
  - Given the neighborhood of solutions $N(s)$, pick one of the solutions and calculate the difference in cost between the old solution and the new neighbor solution.
  - If the new solution is better then accept the new solution. If the old solution is better then generate a **random number** between 0 and 1 and accept it is **less than** the **Energy Magnitude** equation.
  - Loop through *n* iterations of previous two steps and then decrease the temperature according to α. Stop when the solution is accurate within a threshold.

ML & RL Heuristics

- **Genetic (evolutionary) algorithms**: randomized search techniques that simulate some of the processes observed in natural evolution.
- **Ant colony systems (ACS):** a set of cooperating agents called ants cooperate to find good solutions to TSP.
  - (First paper, 1995) **Ant-Q:** was inspired by work on the **ant system** (AS), a distributed algorithm for combinatorial optimization and by the work behind **Q-learning**, a reinforcement learning algorithm.
  - (1997) ACS **outperforms** other nature-inspired algorithms such as simulated annealing and evolutionary computation.

- Github

Why use these more advanced ML heuristics? intuition

- ML heuristics achieve **state-of-the-art** results compared to traditional heuristics. They are also **fast**.
- ML heuristics build upon **solid theoretical models**, whose mathematical properties have been **studied in-depth**.
- RL agents trained on TSP can easily achieve close to optimal results on **other related** NP-hard tasks similar to TSP.

Introducing Neural Networks

- We **map** TSP onto a neural network structure, because the TSP graph is very different from the neural network itself.

- **Hopfield networks:** Github
- **Self-Organizing Maps:** Github
- **Graph Convolutional Networks:** Github

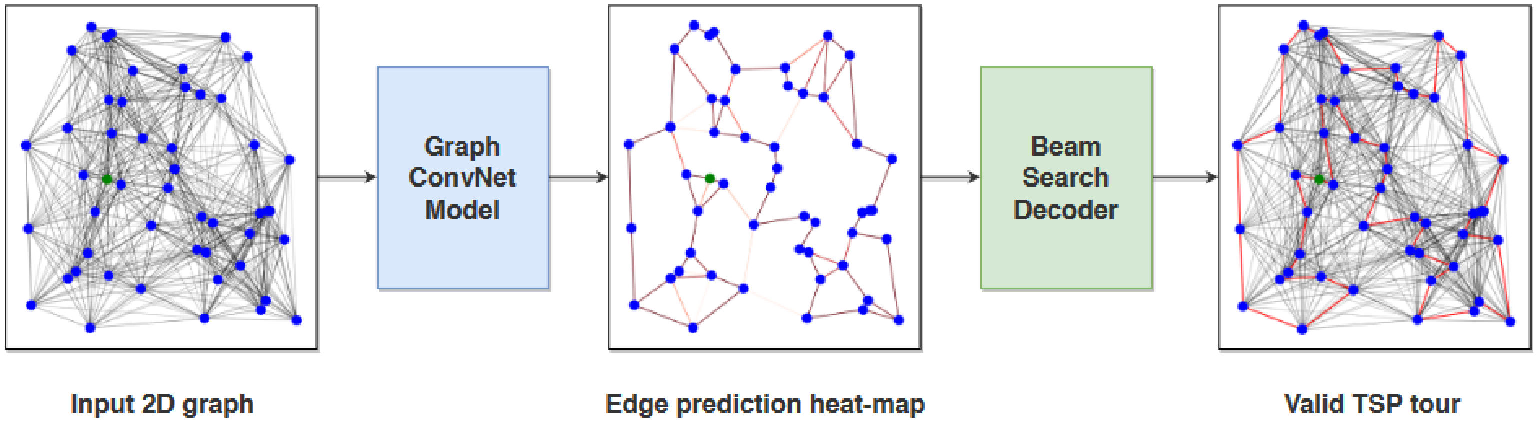
Hopfield Network

- The original Hopfield neural network model is a fully interconnected network of binary units with symmetric connection weights between the units.
- The connection weights are not learned, but are **defined a priori** from problem data.
- Starting from some arbitrarily chosen initial configuration, either feasible or infeasible, the Hopfield network evolves by **updating the activation** of each unit in turn (an activation unit can be turned **on** or **off**).
- The **update rule** of any given unit involves the *activation of the units it is connected to* as well as the *weights* on the connections.
- Via this update process, various configurations are explored until the network settles into a stable configuration.

Self-Organizing Map

- The self-organizing maps are instances of so-called **competitive neural networks**.
- Used in **unsupervised learning** to cluster or classify data. As opposed to supervised models that use cost functions and true labels, unsupervised systems must find **useful correlations** in the input data by themselves.
- A self-organizing map **groups the inputs** on the basis of **similarity** between the input patterns X and the weight vectors Tj.
- In the TSP context, the input patterns are the two-dimensional coordinates of the cities, hence the **Euclidean distance** is used as the similarity measure.

Graph Convolutional Network



- Taking a 2D graph as input, the graph ConvNet outputs an **edge adjacency matrix** denoting the probabilities of edges occurring on the TSP tour.
- This is converted to a valid tour using beam search.
- All components are highly **parallelized** and solutions are produced in a one-shot, non-autoregressive manner.

Novel research using Deep Reinforcement Learning

- **Neural Combinatorial Optimization:** Github
  - using negative tour length as the reward signal, NCO optimizes the parameters of the recurrent network using a **policy gradient method**.
- **Attention based model** trained with **REINFORCE:** Github
  - with greedy rollout baseline to learn heuristics shows competitive results on TSP and other routing problems.
  - with the same hyperparameters, it learns strong heuristics for two variants of the *Vehicle Routing Problem*, the *Orienteering Problem* and the *Prize Collecting TSP.*

References

Bello, I., Pham, H., Le, Q., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. arXiv.

Dorigo, M., & Gambardella, L. (1995). Ant-Q: A reinforcement learning approach to the traveling salesman problem. Machine learning proceedings. Elsevier.

Dorigo, M., & Gambardella, L. (1997). Ant colonies for the travelling salesman problem. Biosystems. Elsevier.

Dorigo, M., & Gambardella, L. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. Transactions on Evolutionary Computation. IEEE.

Joshi, C., Laurent, T., & Bresson, X. (2019). An efficient graph convolutional network technique for the travelling salesman problem. arXiv.

Kool, W., van Hoof, H., & Welling, M. (February 2019). Attention, Learn to Solve Routing Problems! ICLR 2019 Conference Blind Submission. arXiv.

Larrañaga, P., Kuijpers, C., Murga, R., Inza, I., & Dizdarevic, S. (April 1999). Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. Artificial Intelligence Review (pp. 129-170). Springer.

Leung, K.-S., Jin, H.-D., & Xu, Z.-B. (December 2004). An expanding self-organizing neural network for the traveling salesman problem. Neurocomputing. Elsevier.

Moon, C., Kima, J., Choia, G., & Seob, Y. (August 2002). An efficient genetic algorithm for the traveling salesman problem with precedence constraints. European Journal of Operational Research (pp. 606-617). Elsevier.

Potvin, J. (1993). State-of-the-art survey—the traveling salesman problem: A neural network perspective. ORSA Journal on Computing. Informs.

Potvin, J. (June 1996). Genetic algorithms for the traveling salesman problem. Annals of Operations Research (pp. 337-370). Springer.