TASP: Solving TSP using Simulated Annealing

Artur Topal (s5942128)
Juan Diego Sersen Valdeiglesias (s5996848)

October 29, 2025

1 Problem Description

The Traveling Salesman Problem (TSP) is a well-known problem in computing science and combinatorial optimization with numerous real-life applications in areas such as logistics, manufacturing, and network design. TSP can be summarized as a graph problem where one has to find the shortest possible route that visits each vertex exactly once and returns to the starting vertex.

TSP is known to be an NP problem where the brute force approach, which checks all possible routes, has a time complexity of $O(n!)$, making it impractical for large inputs. Consequently, non-deterministic algorithms have been developed to find near-optimal solutions in a reasonable amount of time. This project implements **Simulated Annealing**, a probabilistic optimization method inspired by the physical process, to find a locally optimal solution to TSP.

Input. All parameters to the program can be tweaked via command-line options and the graph can be input randomly or from the standard input. In this report, we use the following run command[1]:

```
tasp –temp=5000 -m40 -Mstd -Asolve < att48.txt
```

where att48.txt is:

```
48
6734 1453
2233 10
5530 1424
401 841
3082 1644
7608 4458
7573 3716
7265 1268
6898 1885
1112 2049
5468 2606
5989 2873
4706 2674
4612 2035
6347 2683
6107 669
7611 5184
7462 3590
7732 4723
5900 3561
4483 3369
6101 1110
5199 2182
1633 2809
```

[1]ATT48 is a dataset with 48 cities from USA.

```
4307 2322
675 1006
7555 4819
7541 3981
3177 756
7352 4506
7545 2801
3245 3305
6426 3173
4608 1198
23 2216
7248 3779
7762 4595
7392 2244
3484 2829
6271 2135
4985 140
1916 1569
7280 4899
7509 3239
10 2676
6807 2993
5185 3258
3023 1942
```

Output. Output must be the cost of the shortest close path that visits each city once. The globally optimal path is 33523. The program gives the number 34978 which is a very good result.

2 Problem Analysis

Simulated annealing is inspired by the process of slowly cooling a heated metal to make it stronger, rather than cooling it rapidly. When cooling is gradual, atoms have an opportunity to move away from suboptimal positions, local minima in energy, and settle into more stable configurations. This idea is applied by allowing for occasional acceptance of worse solutions to escape local minima.

Let C_0 be the initial configuration of n cities, and let its energy E_0 represent the total path cost. We also choose an initial temperature T_0, which controls the probability of accepting worse routes. A new candidate configuration C_{temp} is generated with energy E_{temp}. The next configuration C_1 is chosen as follows:

- If $E_{\text{temp}} < E_0$: the new configuration C_{temp} is always accepted.
- If $E_{\text{temp}} \geq E_0$: the new configuration is accepted with probability

$$P = e^{\frac{E_0 - E_{\text{temp}}}{kT}}$$

  where k is a constant. Otherwise, $C_1 = C_0$.

This probability function means that worse solutions are less likely to be accepted, especially as T decreases. In real annealing, k is Boltzmann's constant, but since this process is simulated, we can simply set $k = 1$ and adjust the starting temperature T_0 to achieve the best performance.

3 Design

3.1 System Overview

The program consists of several self-contained modules such as:

- **Main module:** Handles program initialization, argument parsing, and coordination of all other modules.
- **Graph module:** Abstraction of the graph including creation, distance matrix, and displaying.
- **Path module:** Functions that operate on paths. Each path is represented as an array of indices of vertices in a graph. Among others, the path module contains random permutation generators, cost-computing functions, and helper functions.
- **Solver module:** Implements the Simulated Annealing algorithm.

On the more technical side, the program uses the notion of public and internal headers. The former ones are the headers that provide the interface (functions, data structures, etc.) for the use by other modules. Internal headers declare all the functions and data structures that can be hidden from other modules or the user. For instance, solve.h has one public function (sim_annealing) and one private function (update) used internally. Each function has its own translation unit which allows for easier navigation.

Lastly, each internal header is compiled into a PCH[2] which drastically decreases the compilation time.

3.2 More on the Data Structure Design

Each city is represented by integer coordinates (x_i, y_i) in the range that can be defined by command-line arguments, and each city has an ID equal to its index in the vertices array of the graph. The distance matrix of the graph uses Euclidean distance:

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The distances are precomputed and stored in a two-dimensional matrix before the annealing process begins.

A path is represented as a permutation of city indices $[0, 1, 2, ..., n-1]$. The cost (energy) of a path is calculated as the sum of distances between consecutive cities, returning to the starting city.

3.3 Algorithm Design

The Simulated Annealing algorithm is implemented as an iterative procedure that improves the solution by exploring and accepting new routes according to the function described section 2. The pseudocode for the algorithm is:

```
algorithm Annealing($G(V, D)$, $T$, $\bar{T}$, $m$, $\bar{I}$, $f : \mathbb{R} \to \mathbb{R}$)
input:
    1) graph $G$ with cities $V$ and distance matrix $D$
    2) start and desired temperature $T, \bar{T}$
    3) per-temperature number of iterations $m$
    4) desired no-improvement number $\bar{I}$
    5) the function $f$ that is used to decrease the temperature.
output:
    shortest path $\vec{b}$ that visits each vertex of $G$ once.

    Initialize $\vec{b}$ to starting path.
    Initialize $\vec{c}$ to starting path. (candidate)
    let I = 0 (current number of iterations without improvement)

    while ($T > \bar{T} \lor I < \bar{I}$)
```

[2]precompiled header

```
    let k = 0 (current number of iteration with improvement)
    repeat $m$ times
        $\vec{c} \leftarrow$ shuffle($\vec{c}$)
        if $E(\vec{c}) < E(\vec{b})$ then
            $\vec{b} \leftarrow \vec{c}$
        else
            $\vec{b}$ becomes $\vec{c}$ with probability $P = e^{\frac{E(\vec{b})-E(\vec{c})}{T}}$
        incr $k$ if there $\vec{b}$ changed
    incr $I$ if $k = 0$
    $T \leftarrow f(T)$ (decrease temperature in some way)
  return $\vec{b}$
end
```

Note, that in current implementation of the program the temperature-decreasing function has been hard-coded to $f(T) = 0.99T$. See section 5 for the discussion of possible improvements.

3.4 Algorithm analysis

The algorithm contains two loop statements. The outer while-loop runs until either the desired temperature or the desired number of iterations with no improvement is reached. Suppose the former is the working termination condition. Thus, if the outer loop ran for n iterations, we have

$$T_n = \bar{T} = (f \odot \cdots \odot f)_n(T)$$

which can be further rewritten as another function[3] of n and T:

$$\bar{T} = g_T(n) \text{ where } g_T(n) = (f \odot \cdots \odot f)_n(T)$$

Therefore, $n = g_T^{-1}(\bar{T})$. Now, assume that the second condition is the terminating one. In this case, the number of iterations is nondeterministic as it depends on the randomized choice of the candidate path and the probability. Due to our insufficient knowledge of probability theory, we further assume that the temperature-condition is terminating.

The inner loop runs m times. However, each iteration computes the energy of the path and/or copies $\vec{c}$ into $\vec{b}$. Both are algorithms with linear time complexity which scales with the number of vertices in the graph G. Thus, it is $O(|G|)$ for copy or energy compute. Shuffle has the worst-time complexity of $O(|G|)$ as well but it depends on the random number generated. However, it its worst-time complexity does not become worse than the copy or the energy compute procedures.

Combining the outer and inner loop, the resulting time complexity is

$$O(g_T^{-1}(\bar{T})m|G|) \text{ where } g_T(n) = (f \odot \cdots \odot f)_n(T)$$

Since we hard-coded the function f to be $f(T) = 0.99T$, the time-complexity can be simplified even further:

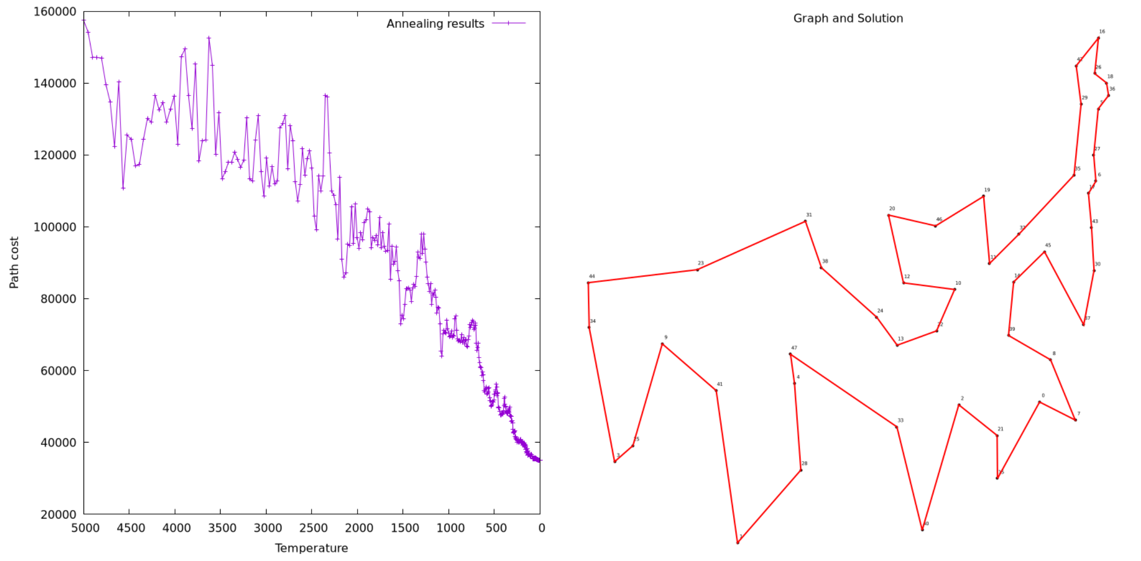
$$g_T(n) = 0.99^n T$$
$$g_T^{-1}(x) = \log_{0.99}(\frac{x}{T})$$

Thus, the final time complexity becomes

$$O(\log_{0.99}(\frac{\bar{T}}{T}) \times m \times |G|)$$

[3]I assume f is a nice function so that the composite function is also nice and thus has an inversion.

4 Evaluation



(a) The plot showing temperature and cost progression during the simulated annealing process.

(b) The plot of the resulting path found by simulated annealing.

Figure 1: ATT48 Results

The steps that simulated annealing performed can be obtained by running **tasp** with the *plot* and *path_vis* action flags. The generated figures are shown in figure 1. The cost does indeed decrease non deterministically and it jumps from one location with local minima to other until eventually reaching the final locally optimal solution at the desired temperature.

5 Improvements

This project had been developed in a time where we had a lot of deadlines and other projects. Therefore, **tasp** was developed quickly and with the *just-enough* mentality. Therefore, not all the desired functionality had been developed. The todo-list can be viewed on the README page of the GitHub of the project [1]. The highlighted possible improvements are listed here:

- Utilize any decreasing temperature function $f : \mathbb{R} \rightarrow \mathbb{R}$ to decrease the temperature instead of hard-coding it.
- Create an action that will find the best parameters of the algorithm by itself.
- Implement the minimum spanning tree algorithm to provide the lower bound for the TSP, which is very useful in determining whether the algorithm reaches a reasonable optimum.
- Use parallel computing to make the algorithm run faster.

6 Conclusion

Concluding, even though TSP is a problem that can be solved using a non-deterministic algorithm in the reasonable time, the found results are still very plausible and reasonable to use in real-life world.

A Program Code

The code and installation guide are available on [1].
`File execute.c`

```
#include "main.ih"

static void (*action_dispatch[])(Config const *, Graph) = {
  &action_plot,                    // ACTION_PLOT
  &action_solve,                   // ACTION_SOLVE
  &action_path,                    // ACTION_PATH
  &action_path_vis,                // ACTION_PATH_VIS
  &action_graph,                   // ACTION_GRAPH
  &action_graph_vis,               // ACTION_GRAPH_VIS
  &action_graph_vis_nol,           // ACTION_GRAPH_VIS_NOL
  &action_distances,               // ACTION_DISTANCES
};

void execute(Config const *config)
{
  srand(config->seed);
  Graph graph = graph_create(config);

  (*action_dispatch[config->action])(config, graph);

  graph_destroy(graph);
}
```

`File action_path.c`

```
#include "main.ih"

void action_path(Config const *config, Graph graph)
{
  size_t *solution = sim_annealing(graph, config->T, config->T_aim,
                                   config->noimpr_aim, config->m, NULL);

  path_display(solution, graph.size);
  free(solution);
}
```

`File action_solve.c`

```
#include "main.ih"

void action_solve(Config const *config, Graph graph)
{
  size_t *solution = sim_annealing(graph, config->T, config->T_aim,
                                   config->noimpr_aim, config->m, NULL);
  printf("%f\n", path_compute(graph, solution));
  free(solution);
}
```

`File main.ih`

```
#include "graph/graph.h"
#include "solve/solve.h"
#include "path/path.h"
```

```
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

typedef enum
{
  ACTION_PLOT,                  // plot temperature - cost
  ACTION_SOLVE,                 // print the distance of the found path
  ACTION_PATH,                  // print the resulting path
  ACTION_PATH_VIS,              // visualize the resulting path
  ACTION_GRAPH,                 // print the resulting graph
  ACTION_GRAPH_VIS,             // visualize the resulting graph (with labels)
  ACTION_GRAPH_VIS_NOL,         // visualize the resulting graph (no labels)
  ACTION_DISTANCES,             // print the distances matrix of the graph
  ACTION_INVALID,
} Action;


typedef enum
{
  MODE_RANDOM,                  // generate cities randomly
  MODE_STDIN,                   // input cities via stdin
  MODE_INVALID,
} Mode;

typedef struct
{
  int ok;                       // if parsing is successful
  size_t argc;                  // number of variadic arguments (...args)
  char **argv;                  // variadic arguments (...args)
  Mode mode;                    // input mode for vertices
  Action action;                // action to perform
  size_t seed;                  // seed for random number generator
  size_t size;                  // #vertices to randomly generate
  int min;                      // min possible coordinate
  int max;                      // max possible coordinate
  float T;                      // temperature for SA
  float T_aim;                  // desired temperature for SA
  size_t m;                     // m-parameter for SA
  size_t noimpr_aim;            // desired #iterations with no improvement
  char *output;                 // .png file where vis will be displayed
} Config;

void usage(char const *progname);
Config parse_arguments(int argc, char **argv);

void execute(Config const *config);

Graph graph_create(Config const *config);
void action_plot(Config const *config, Graph graph);
void action_solve(Config const *config, Graph graph);
void action_path(Config const *config, Graph graph);
void action_path_vis(Config const *config, Graph graph);
void action_graph(Config const *config, Graph graph);
void action_graph_vis(Config const *config, Graph graph);
void action_graph_vis_nol(Config const *config, Graph graph);
void action_distances(Config const *config, Graph graph);

void run_gnuplot(char const *script, char const *outfile);
```

File `run_gnuplot.c`

```
#include "main.ih"

void run_gnuplot(char const *script, char const *outfile)
{
  char cmd[300];                // 300 is enough for the command in the next line
  sprintf(cmd, "gnuplot -e \"outfile='%s'\" %s/%s && rm data",
          outfile, SCRDIR, script);
  system(cmd);
}
```

File `action_path_vis.c`

```
#include "main.ih"

void action_path_vis(Config const *config, Graph graph)
{
  FILE *out = fopen("data", "w");
  if (!out)
  {
    printf("Unable to open the stream.\n");
    exit(1);
  }

  size_t *solution = sim_annealing(graph, config->T, config->T_aim,
                                   config->noimpr_aim, config->m, NULL);

                                // write the graph coordintes into out
  for (size_t i = 0; i != graph.size; ++i)
    fprintf(out, "%d %d %zu\n", graph.data[i].x, graph.data[i].y, i);
                                // write the path into out
  fprintf(out, "\n");
  for (size_t i = 0; i != graph.size + 1; ++i)
    fprintf(out, "%zu\n", solution[i % graph.size]); // include the start vertex

  if (out)
    fclose(out);
  free(solution);

  run_gnuplot("solve.gnu", config->output);
}
```

File `solve.ih`

```
#include "solve.h"

#include "../path/path.h"

#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>

// Update the best path with candidate, and return the cost of the new path.
// Update conditions:
// * Cost(candidate) < Cost(best)  => always accept
// * Cost(candidate) >= Cost(best) => accept with prob = exp(-deltaCost / T)
float update(Graph graph, size_t *best, size_t *candidate, float T);
```

File `sim_annealing.c`

```
#include "solve.ih"

size_t *sim_annealing(Graph graph, float T, float T_aim, size_t noimpr_aim,
                      size_t m, FILE *out)
{
  size_t *best      = path_init(graph.size);
  size_t *candidate = path_init(graph.size);

  float best_cost = path_compute(graph, best);
  size_t noimpr = 0;           // #iterations with no improvement

  while (T > T_aim && noimpr < noimpr_aim)
  {
    if (out)
      fprintf(out, "%f %f\n", T, best_cost);

    size_t impr = 0;           // #iterations with improvement for T

    for (size_t i = 0; i != m; ++i) // repeat m times for the same T
    {
      path_copy(candidate, best, graph.size);
      path_shuffle(candidate, graph.size);

      float const old_best_cost = best_cost;
      best_cost = update(graph, best, candidate, T);

      impr += best_cost < old_best_cost;
    }

    noimpr = (impr) ? (0) : (noimpr + 1);
    T *= 0.99;
  }

  free(candidate);
  return best;
}
```

File `solve.h`

```
#ifndef INCLUDED_SOLVE_
#define INCLUDED_SOLVE_

#include "../graph/graph.h"

#include <stdio.h>
#include <stddef.h>
```

```
// Solve TSP using Simulated Annealing.
// Parameters:
// - graph: the graph being solved.
// - T: initial temperature.
// - T_aim: desired temperature.
// - noimpr_aim: desired #iterations without improvement.
// - m: #iterations to be performed per one temperature value.
// - out: stream to store all (temp, cost) pairs for plotting.
// Returns: the best path
size_t *sim_annealing(Graph graph, float T, float T_aim, size_t noimpr_aim,
                      size_t m, FILE *out);

#endif
```

File update.c

```
#include "solve.ih"

float update(Graph graph, size_t *best, size_t *candidate, float T)
{
  float const                     // compute delta cost of best and candidatest
    best_cost      = path_compute(graph, best),
    candidate_cost = path_compute(graph, candidate),
    delta_cost     = candidate_cost - best_cost;

  float cost = best_cost;
  int accept = 1;                 // whether to accept the candidate

  if (delta_cost > 0)             // sometimes accept worse candidate
  {
    float const
      prob   = expf(-delta_cost / T),
      choice = (float)rand() / RAND_MAX;

    accept = choice < prob;
  }

  if (accept)
  {
    path_copy(best, candidate, graph.size);
    cost = candidate_cost;
  }

  return cost;
}
```

File path.h

```
#ifndef INCLUDED_PATH_
#define INCLUDED_PATH_

#include "../graph/graph.h"

#include <stddef.h>


size_t *path_init(size_t size);
```

```
void path_copy(size_t *dest, size_t *src, size_t size);

float path_compute(Graph graph, size_t *path);  // computes path cost

int path_shuffle(size_t *path, size_t size);

void path_shuffle_display(int method); // display the path shuffle method
void path_display(size_t *path, size_t size);


#endif
```

File path_display.c

```
#include "path.ih"

void path_display(size_t *path, size_t size)
{
  printf("[ ");
  for (size_t i = 0; i != size + 1; ++i)
    printf("%zu ", path[i % size]);
  printf("]\n");
}
```

File path_shuffle_invert.c

```
#include "path.ih"

void path_shuffle_invert(size_t *path, size_t size)
{
  Pair pair = get_random_pair(size);
                                // invert segment [idx1..idx2]
  size_t left = pair.idx1;
  size_t right = pair.idx2;
  while (left < right)
    swap(path + left++, path + right--);
}
```

File path_shuffle_display.c

```
#include "path.ih"

void path_shuffle_display(int method)
{
  char const *shuffle_str;

  switch (method)
  {
  case 0:
    shuffle_str = "SWAP";
    break;
  case 1:
    shuffle_str = "INVERT";
    break;
  case 2:
    shuffle_str = "SHIFT";
    break;
  default:
    shuffle_str = "INVALID";
    break;
  }

  printf("%s", shuffle_str);
}
```

File path_init.c

```
#include "path.ih"

size_t *path_init(size_t size)
{
  size_t *path = malloc(size * sizeof(size_t));
                                // initial path: 0 1 2 ... n-2 n-1
  for (size_t i = 0; i != size; ++i)
    path[i] = i;

  return path;
}
```

File path_compute.c

```
#include "path.ih"

float path_compute(Graph graph, size_t *path)
{
  float cost = 0;

  for (size_t i = 0; i != graph.size; ++i)
  {
    size_t from = path[i];
    size_t to = path[(i + 1) % graph.size];
    cost += graph.distance[from * graph.size + to];
  }

  return cost;
}
```

File path_shuffle_shift.c

```
#include "path.ih"

void path_shuffle_shift(size_t *path, size_t size)
{
  size_t *tmp = malloc(size * sizeof(size_t));

                                // perform cyclic shift and store in tmp
  for (size_t i = 0; i != size; ++i)
    tmp[i] = path[(i + 1) % size];

  path_copy(path, tmp, size);   // copy tmp back into path
  free(tmp);
}
```

File path_shuffle_swap.c

```
#include "path.ih"
```

```
void path_shuffle_swap(size_t *path, size_t size)
{
  Pair pair = get_random_pair(size);
  swap(path + pair.idx1, path + pair.idx2);
}
```

File `swap.c`

```
#include "path.ih"

void swap(size_t *a, size_t *b)
{
  size_t tmp = *a;
  *a = *b;
  *b = tmp;
}
```

File `path_shuffle.c`

```
#include "path.ih"

int path_shuffle(size_t *path, size_t size)
{                              // we have 3 method to shuffle
  static void (*shuffle[3])(size_t *, size_t) = {
    &path_shuffle_swap,
    &path_shuffle_invert,
    &path_shuffle_shift,
  };

  int const shuffle_method = rand() % 3;
  (*shuffle[shuffle_method])(path, size);

  return shuffle_method;
}
```

File `path_copy.c`

```
#include "path.ih"

void path_copy(size_t *dest, size_t *src, size_t size)
{
  for (size_t i = 0; i != size; ++i)
    dest[i] = src[i];
}
```

File `path.ih`

```
#include "path.h"

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

typedef struct
{
  size_t idx1;
  size_t idx2;
} Pair;
```

```
void swap(size_t *a, size_t *b);  // swap two size_t numbers

Pair get_random_pair(size_t size);

void path_shuffle_invert(size_t *path, size_t size);
void path_shuffle_swap(size_t *path, size_t size);
void path_shuffle_shift(size_t *path, size_t size);
```

File get_random_pair.c

```
#include "path.ih"

Pair get_random_pair(size_t size)
{                                           // 0 <= idx1 <= idx2 < n
  size_t const idx1 = rand() % size;
  size_t const idx2 = rand() % size;

  return (Pair){
    .idx1 = (idx1 < idx2) ? idx1 : idx2,
    .idx2 = (idx1 < idx2) ? idx2 : idx1,
  };
}
```

File usage.c

```
#include "main.ih"

                                // see parse_arguments.c for long_options
static char const s_usage_info[] = "Usage: %s [OPTIONS] -A action
\n\
Options: <optional>
\n\
  --help       (-h)       - displays this message.
\n\
  --seed       (-s) num  - seed for RNG (DEFAULT: 69)
\n\
  --size       (-S) num  - #vertices to generate (DEFAULT: 1000)
\n\
  --min        (-n) num  - minimum coord (DEFAULT: -1000)
\n\
  --max        (-x) num  - maximum coord (DEFAULT: 1000)
\n\
  --temp       (-t) num  - initial temperature (DEFAULT: 1000.0f)
\n\
  --tempAim    (-T) num  - desired temperature (DEFAULT: 0.4f)
\n\
  --mParam     (-m) num  - m-parameter (DEFAULT: 10)
\n\
  --noimprAim (-i) num  - max no-improvement #iterations (DEFAULT: 1000) \n\
  --output     (-o) path - .png file where the vis will be displayed
\n\
  --mode       (-M) str  - specifies the mode of city retrieval
\n\
    Where str:
\n\
      rand    - generate random cities (DEFAULT)
\n\
      std     - input cities via stdin until STOP (fmt: <id, x, y>)
\n\
Action: <required>
\n\
  --action (-A) str
\n\
    Where str:
\n\
      plot          - plot the cost versus temperature
\n\
      solve         - print the distance of the found path
\n\
      path          - print the found path
\n\
      path_vis      - visualize the found path
\n\
      graph         - print the graph being analyzed (DEFAULT)
\n\
      graph_vis     - visualize the graph being analyzed (with labels)
\n\
      graph_vis_nol - visualize the graph being analyzed (no labels)
\n\
      distances     - print the distance matrix for the graph
\n";

void usage(char const *progname)
{
  printf(s_usage_info, progname);
}
```

File main.c

```
#include "main.ih"

int main(int argc, char **argv)
{
  Config config = parse_arguments(argc, argv);
  if (!config.ok)
  {
    usage(argv[0]);
    exit(1);
  }

  execute(&config);
  return 0;
}
```

File action_graph.c

```
#include "main.ih"

void action_graph(Config const *config, Graph graph)
{
  graph_display(graph);
}
```

File action_plot.c

```
#include "main.ih"

void action_plot(Config const *config, Graph graph)
{
  FILE *out = fopen("data", "w");
  if (!out)
  {
    printf("Unable to open the stream.\n");
    exit(1);
  }

  free(sim_annealing(graph, config->T, config->T_aim,
                     config->noimpr_aim, config->m, out));
  run_gnuplot("plot.gnu", config->output);
}
```

File `parse_arguments.c`

```
#include "main.ih"

static Mode opt_to_mode(const char *str)
{
  if (strcmp(str, "rand") == 0)
    return MODE_RANDOM;
  if (strcmp(str, "std") == 0)
    return MODE_STDIN;

  return MODE_INVALID;
}

static Action opt_to_action(const char *str)
{
  if (strcmp(str, "plot") == 0)
    return ACTION_PLOT;
  if (strcmp(str, "solve") == 0)
    return ACTION_SOLVE;
  if (strcmp(str, "path") == 0)
    return ACTION_PATH;
  if (strcmp(str, "path_vis") == 0)
    return ACTION_PATH_VIS;
  if (strcmp(str, "graph") == 0)
    return ACTION_GRAPH;
  if (strcmp(str, "graph_vis") == 0)
    return ACTION_GRAPH_VIS;
  if (strcmp(str, "graph_vis_nol") == 0)
    return ACTION_GRAPH_VIS_NOL;
  if (strcmp(str, "distances") == 0)
    return ACTION_DISTANCES;

  return ACTION_INVALID;
}

static Config s_config = {        // default configuration
  .ok          = 1,
  .mode        = MODE_RANDOM,
  .action      = ACTION_GRAPH,
  .seed       = 69,
  .size       = 50,
  .min        = -100,
  .max        = 100,
  .T          = 1000.0f,
  .T_aim      = 0.4f,
  .m          = 10,
  .noimpr_aim = 1000,
  .output     = "result.png",
};

static char const short_options[] = "hs:S:n:x:t:T:m:i:M:o:A:";
static struct option long_options[] = {
  {"help",      no_argument,       0, 'h'},
  {"seed",      required_argument, 0, 's'},
  {"size",      required_argument, 0, 'S'},
  {"min",       required_argument, 0, 'n'},
  {"max",       required_argument, 0, 'x'},
  {"temp",      required_argument, 0, 't'},
  {"tempAim",   required_argument, 0, 'T'},
  {"mParam",    required_argument, 0, 'm'},
  {"noimprAim", required_argument, 0, 'i'},
  {"mode",      required_argument, 0, 'M'},
  {"output",    required_argument, 0, 'o'},
  {"action",    required_argument, 0, 'A'},
  {0, 0, 0, 0}
};

Config parse_arguments(int argc, char **argv)
{
  if (argc == 1)                 // no arguments provided, default config
    return s_config;

  while (1)                      // while we can parse
  {
    errno = 0;                   // for strtoul error checking
    int option_index;
    int short_opt = getopt_long(argc, argv, short_options, long_options,
                                &option_index);

    if (short_opt == -1)         // no more options
      break;

    switch (short_opt)
    {
    case 'h':
      usage(argv[0]);
      exit(0);
      break;
    case 's':
      s_config.seed = strtoul(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 'S':
      s_config.size = strtoul(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 'n':
      s_config.min = strtol(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 'x':
      s_config.max = strtol(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 't':
      s_config.T = strtof(optarg, NULL);
      if (errno)
        s_config.ok = 0;
      break;
    case 'T':
      s_config.T_aim = strtof(optarg, NULL);
      if (errno)
        s_config.ok = 0;
      break;
    case 'm':
      s_config.m = strtoul(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 'i':
      s_config.noimpr_aim = strtoul(optarg, NULL, 10);
      if (errno)
        s_config.ok = 0;
      break;
    case 'M':
      s_config.mode = opt_to_mode(optarg);
      if (s_config.mode == MODE_INVALID)
        s_config.ok = 0;
      break;
    case 'o':
      s_config.output = strdup(optarg);
      break;
    case 'A':
      s_config.action = opt_to_action(optarg);
      if (s_config.action == ACTION_INVALID)
        s_config.ok = 0;
      break;
    default:
      printf("?? getopt returned character code 0%o ??\n", short_opt);
    }
  }

  return s_config;
}
```

File action_distances.c

```
#include "main.ih"

void action_distances(Config const *config, Graph graph)
{
  graph_distances_display(graph);
}
```

File `action_graph_vis.c`

```
#include "main.ih"

void action_graph_vis(Config const *config, Graph graph)
{
  FILE *out = fopen("data", "w");
  if (!out)
  {
    printf("Unable to open the stream.\n");
    exit(1);
  }
                                // write the graph coordintes into out
  for (size_t i = 0; i != graph.size; ++i)
    fprintf(out, "%d %d %zu\n", graph.data[i].x, graph.data[i].y, i);

  if (out)
    fclose(out);

  run_gnuplot("graph.gnu", config->output);
}
```

File `graph_create.c`

```
#include "main.ih"

Graph graph_create(Config const *config)
{
  Graph graph;

  switch (config->mode)
  {
  case MODE_RANDOM:
    graph = graph_create_rand(config->size, config->min, config->max);
    break;
  case MODE_STDIN:
    graph = graph_create_stdin();
    break;
  default:                      // shouldn't happen
  }

  return graph;
}
```

File `action_graph_vis_nol.c`

```
#include "main.ih"

void action_graph_vis_nol(Config const *config, Graph graph)
{
  FILE *out = fopen("data", "w");
  if (!out)
  {
    printf("Unable to open the stream.\n");
    exit(1);
  }
                                  // write the graph coordintes into out
  for (size_t i = 0; i != graph.size; ++i)
    fprintf(out, "%d %d\n", graph.data[i].x, graph.data[i].y);

  if (out)
    fclose(out);

  run_gnuplot("graph_nol.gnu", config->output);
}
```

File **graph.ih**

```
#include "graph.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void precompute_distances(Graph *graph);
```

File **graph_create_rand.c**

```
#include "graph.ih"

Graph graph_create_rand(size_t size, int min, int max)
{
  Graph graph = {
    .size = size,
    .data = malloc(size * sizeof(Vertex)),
    .distance = (void *)0,
  };

  for (size_t i = 0; i != size; ++i)
  {
    graph.data[i].x = min + rand() % (max - min + 1);
    graph.data[i].y = min + rand() % (max - min + 1);
  }

  precompute_distances(&graph);
  return graph;
}
```

File **graph_create_stdin.c**

```
#include "graph.ih"

Graph graph_create_stdin(void)
{
  Graph graph = {0};
  scanf("%zu", &graph.size);     // get graph size

                                  // get graph vertices
  graph.data = malloc(graph.size * sizeof(Vertex));
  for (size_t i = 0; i != graph.size; ++i)
    scanf("%d %d", &graph.data[i].x, &graph.data[i].y);

  precompute_distances(&graph);
  return graph;
}
```

File `graph_display.c`

```
#include "graph.ih"

void graph_display(Graph graph)
{
  printf("Graph.\n");
  printf("========================================\n");
  printf("Size = %zu\n", graph.size);

  printf("Cities:\n");
  for (size_t i = 0; i != graph.size; ++i)
    printf("%zu: (x=%d, y=%d)\n", i, graph.data[i].x, graph.data[i].y);
  printf("========================================\n");
}
```

File `graph_distances_display.c`

```
#include "graph.ih"

void graph_distances_display(Graph graph)
{
  printf("Distances:\n");
  printf("========================================\n");
  for (size_t i = 0; i != graph.size; ++i)
  {
    for (size_t j = 0; j != graph.size; ++j)
      printf("%.2f ", graph.distance[i * graph.size + j]);
    printf("\n");
  }
  printf("========================================\n");
}
```

File `precompute_distances.c`

```
#include "graph.ih"

void precompute_distances(Graph *graph)
{
  graph->distance = malloc(graph->size * graph->size * sizeof(float));

  for (size_t i = 0; i != graph->size; ++i)
  {
    graph->distance[i * graph->size + i] = 0;  // dist(A, A) = 0
    for (size_t j = i + 1; j != graph->size; ++j)
    {
      int dx = graph->data[i].x - graph->data[j].x;
      int dy = graph->data[i].y - graph->data[j].y;
      float dist = sqrtf(dx * dx + dy * dy);

      graph->distance[i * graph->size + j] = // dist(A, B) = dist(B, A)
```

```
        graph->distance[j * graph->size + i] = dist;
      }
    }
}
```

File `graph.h`

```
#ifndef INCLUDED_GRAPH_
#define INCLUDED_GRAPH_

#include <stddef.h>

typedef struct
{
    int x, y;
} Vertex;

typedef struct
{
    size_t size;
    Vertex *data;
    float *distance;
} Graph;

Graph graph_create_rand(size_t size, int min, int max);
Graph graph_create_stdin(void);

void graph_destroy(Graph graph);           // destroy the graph
void graph_display(Graph graph);           // display the graph
void graph_distances_display(Graph graph); // display the distances matrix

#endif
```

File `graph.destroy.c`

```
#include "graph.ih"

void graph_destroy(Graph graph)
{
  free(graph.data);
  free(graph.distance);
}
```

References

[1] Artur Topal (2025) TASP, `https://github.com/horki-at/tasp`