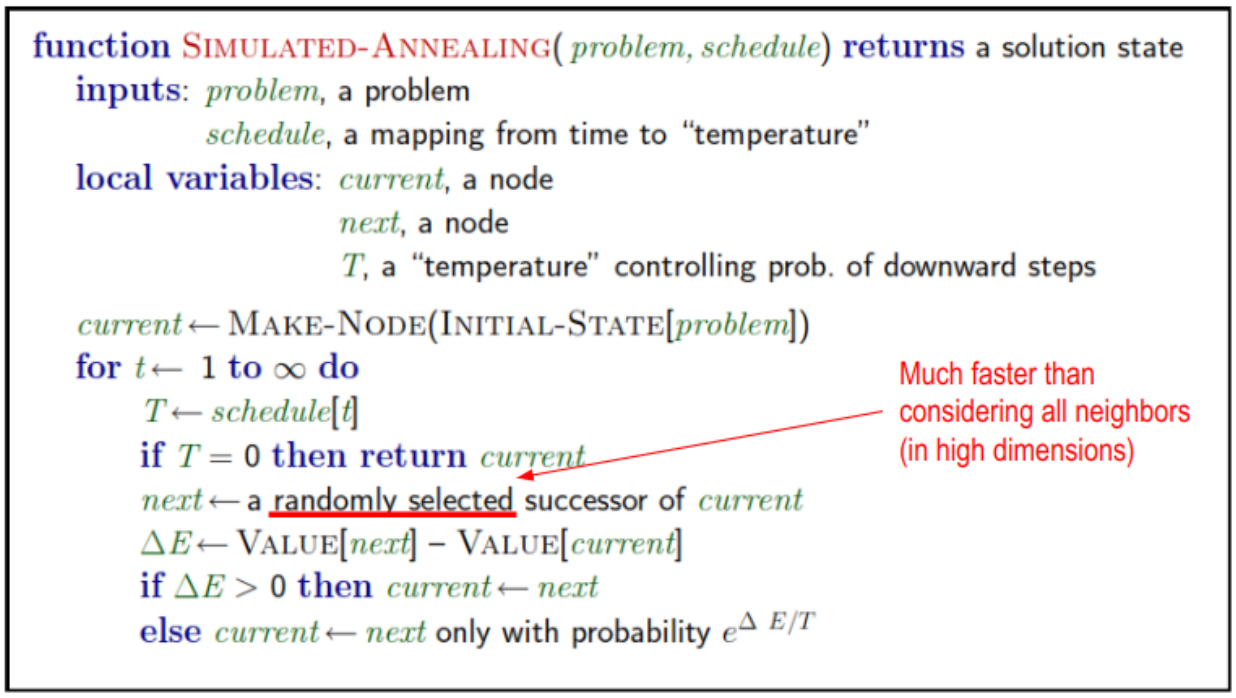
Lab 20: Search for Optimization

This lab covers some local search methods for approximately solving non-convex optimization problems. These problems are usually infeasible to solve globally, so local search methods provide solutions try to find good local minima via making local adjustments to a guess solution. The different methods are just different ways to perturb the local solution.

1. **Simulated annealing** to solve the travelling salesman (delivery robot) problem. Suppose you work for Amazon and need to make deliveries to N households. Assuming you can start the delivery at any household, find the path that leads to the minimum total distance travelled.

   The solution is the order in which we visit the locations, represented as a vector of indices `using Assignment = std::vector<int>` ; note that this already makes our problem non-convex since the space over which we're optimizing is non-convex. Intuitively, for the solution space to be convex, any value in between two valid solutions must also be valid. Since non-integer values aren't valid, this makes the solution set non-convex.

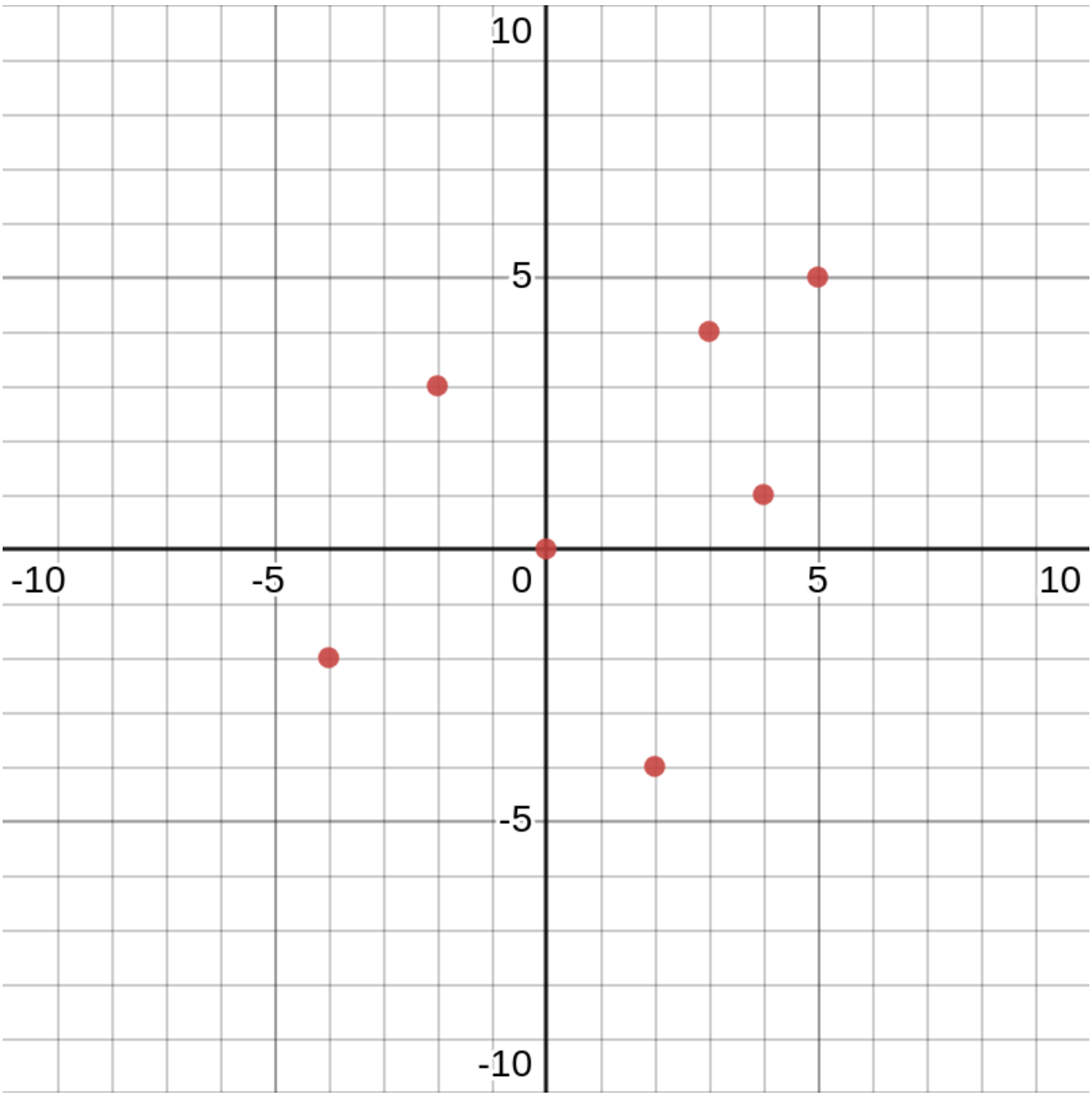


First set up the problem:

```
#include <iostream>
#include <vector>
```

```
#include <cmath>

class Point {
public:
    Point(double x, double y) : x(x), y(y) {}
    double x,y;
};

using Problem = std::vector<Point>;
using Assignment = std::vector<int>;

class Solution {
public:
    Solution(size_t N) {
        // default solution; each solution must be valid
        for (size_t i = 0; i < N; ++i) {
            order.push_back(i);
        }
    }
    // the order in which the points are visited; note that we never need to visit a node twice
    // therefore the assignment is a permutation of 0,1,...,N-1
    Assignment order;
    double cost;
};

double distance(const Point& a, const Point& b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return std::sqrt(dx*dx + dy*dy);
}

int main() {
    Problem prob;
    prob.emplace_back(0, 0);
    prob.emplace_back(5, 5);
    prob.emplace_back(-2, 3);
    prob.emplace_back(4, 1);
    prob.emplace_back(3, 4);
    prob.emplace_back(-4, -2);
    prob.emplace_back(2, -4);
}
```

See https://www.desmos.com/calculator/hfg7hbccj3 for the visual plot of the 2D points



For simulated annealing, we need to be able to quickly evaluate the cost of any solution, and also perturb the solution to a valid neighbor. Note that each solution proposal has to be valid, and in most cases this may not be so trivial, but because our solution is an ordering, it is a permutation of the indices. Swapping two indices from a valid solution always results in a valid solution. Implement the two functions below (can use `rand() % N` to get an integer between 0 and N-1):

```
double scoreAssignment(const Problem& points, const Assignment& order) {
}

Assignment perturbAssignment(Assignment order) {
    // randomly swap the order of two items
```

```
    // fine if sometimes you swap an element with itself
}
```

Now implement the simulated annealing algorithm; our cooling schedule will be a simple multiplicative one (delta < 1):

```
t <- t * delta;
```

```
class SimulatedAnnealingParameters {
public:
    double maxTemperature;
    double deltaTemperature;
    double finishTemperature;
};
Solution simulatedAnnealing(const Problem& problem, Solution sol, SimulatedAnnealingParameters params) {
for (double t = params.maxTemperature; t > params.finishTemperature; t *= params.deltaTemperature) {
        // --- Your code here
        std::cout << "cur cost " << sol.cost << " sample cost " << cost << " prob " << prob << std::endl;
        // ---
    }
    return sol;
}
```

And call it in the updated main

```
int main() {
    Problem prob;
    prob.emplace_back(0, 0);
    prob.emplace_back(5, 5);
    prob.emplace_back(-2, 3);
    prob.emplace_back(4, 1);
    prob.emplace_back(3, 4);
    prob.emplace_back(-4, -2);
    prob.emplace_back(2, -4);

    Solution sol(prob.size());
    // solution of default assignment
    sol.cost = scoreAssignment(prob, sol.order);
    std::cout << sol.cost << std::endl;

    SimulatedAnnealingParameters params {10, 0.99, 0.001};
    auto saSolution = simulatedAnnealing(prob, sol, params);
    std::cout << "simulated annealing solution\ncost " << saSolution.cost << std::endl;
    for (const auto& i : saSolution.order) {
        std::cout << i << ' ';
    }
    std::cout << std::endl;
    for (const auto& i : saSolution.order) {
        std::cout << prob[i].x << ' ' << prob[i].y << std::endl;
    }
}
```

Run it multiple times with different parameters to see what kind of results you get.

2. **Genetic algorithms**. There are many variants of genetic algorithms, but many have the same high level structure. Compared to simulated annealing, they optimize populations of solutions at a time, rather than a single one.

For this problem, let us attempt the LP problem from homework 5:

Name	Price per hour	heavy lifting	materials transport	earth moving	concrete pouring	brick laying
SpiderBot P8	$75	1.6	3.5	0.1	2.3	6.1
Gigantimus Maximus	$128	7.2	2.1	7.1	3.2	0.1
VersaDroid X17	$70	3.7	3.2	2.9	3.4	4.9
HedonismBot	$34	0.1	0.15	0.1	0.15	0.1

Instead of constraining the solution to satisfy the hours required (as genetic algorithms don't work well with hard constraints), we instead add a penalty to the objective function. We can penalize the objective function by $200 for each hour of each unfulfilled task. Since this is above $128 (the highest per-hour cost of any robot) this ensures that the optimal solution is still the same as the constrained problem.

We first need to define the fitness function (similar to `scoreAssignment` from above). **Hint**: I recommend using Eigen matrices to represent the cost and constraint penalty; think of `Ax - b`.

```
#include <eigen3/Eigen/Eigen>

using Assignment = Eigen::Matrix<double, 4, 1>;
class Problem {
public:
    Eigen::Matrix<double, 5, 1> requiredHours;
    Eigen::Matrix<double, 4, 1> costPerRobot;
    Eigen::Matrix<double, 5, 4> robotCapabilities;
};

double penaltyPerHour = 200;
double fitness(const Problem& prob, const Assignment& hours) {
  // --- Your code here
  // ---
}
```

We can first test whether this is working:

```
#include <iostream>
int main() {
    Problem prob;
    prob.requiredHours << 10, 25, 15, 5, 20;
    prob.costPerRobot << 75, 128, 70, 34;
    // --- Your code here (populate up robotCapabilities)
    // ---
```

```
    Assignment skipAllWork;
    skipAllWork << 0, 0, 0, 0;
    std::cout << fitness(prob, skipAllWork) << std::endl;

    Assignment test;
    test << 2.5, 1.5, 2.0, 1.0;
    std::cout << fitness(prob, test) << std::endl;
}
```

You should see that the test assignment has a cost of `1863.5` (cost 553.5, penalty 1310).

To begin, we need to randomly generate a population. Let us work with a population of size 1000, and randomly populate the hours for each assignment from 0 to 3 (use `rand() / RAND_MAX`).

```
class Solution {
public:
    Assignment hours;
    double cost;
};
bool compareSolution(const Solution& a, const Solution& b) {
    return a.cost < b.cost;
}
using Population = std::vector<Solution>;

int main() {
    // ---
    const auto populationSize = 1000;
    Population population(populationSize);
    double hourMax = 3;
    for (int i = 0; i < populationSize; ++i) {
        population[i].hours << (double)rand() / RAND_MAX * 3,
            (double)rand() / RAND_MAX * 3, (double)rand() / RAND_MAX * 3,
            (double)rand() / RAND_MAX * 3;
        population[i].cost = fitness(prob, population[i].hours);
    }

}
```

We then evaluate the fitness of each assignment and select the top 10% to generate the next population. Specifically, we will implement a very simple form of crossover and mutation. We select 2 random solutions then choose the first 2 hours from the first selected one and the last 2 hours from the second selected one. We then mutate the solution by randomly adding hours between -0.1 and 0.1 for each robot (but never falling below 0). Add the following to main:

```
    // 100 iterations of evoluation
    for (int k = 0; k < 100; ++k) {
        std::sort(population.begin(), population.end(), compareSolution);
        int cutoffIndex = populationSize / 10;
        Population nextPopulation(populationSize);
        for (int i = 0; i < populationSize; ++i) {
            const Solution& a = population[rand() % cutoffIndex];
            const Solution& b = population[rand() % cutoffIndex];
```

```
            nextPopulation[i] = crossOver(a, b);
            mutate(nextPopulation[i]);

            nextPopulation[i].cost = fitness(prob, nextPopulation[i].hours);
        }
        population = nextPopulation;
    }
```

Where you need to implement

```
Solution crossOver(const Solution& a, const Solution& b) {
  //your code here
}

void mutate(Solution& a) {
  //your code here
}
```

We can then examine the best solution in the population

```
    std::sort(population.begin(), population.end(), compareSolution);
    std::cout << "best in population\n" << population[0].hours << std::endl;
    std::cout << "cost: " << population[0].cost << std::endl;
```

I recommend you add optional verbose prints to `fitness` to see how many hours are fulfilled / missing for each task, and the breakdown of the cost into robot hiring cost and the penalty. The optimal solution should have 0 penalty.