Solving Traveling Salesman problem using simmulated annealing

Michel Bje Randahl Nielsen (s093481)

July 1, 2015

1 About the implementations...

All implementations are done in F# scripts but uses functionality offered by the R programming environment. The scripts utilizes the type provider functionality which F# provides, to invoke functions in R.

F# project page: http://fsharp.org/

F# RProvider project page: https://bluemountaincapital.github.io/FSharpRProvider/

All scripts has been devloped inside visual studio with F# 3.1, but is only dependend on the existance of .Net 4.5 and a F# 3.1 instalation to be run. A full project has been attached to these hand ins and contains all of the scripts and nescesary dll's.

Note that all scripts must run following code in the beginning in order to function, but this piece of code has been omitted in these reports.

```
#r @"..\packages\R.NET.Community.1.5.16\lib\net40\RDotNet.dll"
#r @"..\packages\R.NET.Community.FSharp.0.1.9\lib\net40\RDotNet.FSharp.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.dll"
#r @"..\packages\RProvider.1.1.8\lib\net40\RProvider.Runtime.dll"

open RDotNet
open RProvider
open RProvider.stats
open RProvider.graphics
open RProvider.mvtnorm
open RProvider.MASS
open System
```

2 About the traveling salesman problem

The Traveling salesman problem (TSP) is about finding the shortest route for visiting a set of positions or cities. TSP appears in a broad range of domains and many applications rely on fast and optimal solutions for TSP. But the problem is NP complete and an exact algorithm to solve the problem efficiently and completely has yet to be found. There are however many proposals for different algorithms that finds suboptimal solutions in a timely manner.

In this small project I implement a simulated annealing algorithm and a genetic algorithm, for comparison, to solve TSP. Finally is the simulated annealing algorithm used to fine tune the parameters of the genetic algorithm. The motivation for doing this, is not so much to find good parameters for genetic algorithms, but more to explore the parameter fine tuning capabilities of the simulated annealing algoritm. Simulated annealing could also be used to train neural networks or fine tune parameters in other machine learning algorithms. TSP serves as an exelent toy example for exploring parameter tuning as the problem itself is very difficult to solve efficiently and can be verified visually very easy. A good heuristic for identifying a good solution, is that it should not have any intersections.

2.1 The implementations

The implementations in this report, utilizes functionality defined in a 'Utils' script. This script can be found in the appendix. Furthermore has the project containing the code been attached to this hand in.

3 Genetic algorithm

The genetic algorithm does not relate directly to stochastic simulation, but it is a stochastic algorithm. The algorithm functions by maintaining a population of candidate solutions, which in the case of the TSP is permutations of the cities to travel. In each iteration of the algorithm, candidate solutions are scored and the best of them are combined and mutated and added to the population. The members in the population with a bad score will eventually be kicked out of the population to make room for the better performing members.

An implementation of the algorithm is provided below.

```
type Configuration = {
    PopulationCount: int //number of population members (random gene
       permutations)
    FlatCount: int //algorithm will terminate when it has seen the same
       value this many times
    Mutationchance: float //chance of mutation
    MatePercent: float //part of the population where mother candidates are
       chosen
    CrossOverPercent: float //percentage of genes to cross over when mating
}

//genetic algorithm which takes a set of 'genes' and a configuration as
   argument
//and attempts to find an optimal permutation of the genes
let GeneticAlgorithm (config: Configuration) (genes: Position []) =
    //function that combines two members of the population
    let crossover (mother: Position []) (father: Position []) =
        let cut_length = float(mother.Length) * config.CrossOverPercent
                         |> Math.Ceiling |> int

        //function that swaps indexes to mutate a population member
        let mutate (genes: Position []) =
            let swap_index1 = random() * (float genes.Length) |> int
            let swap_index2 =
                random() * (float genes.Length)
                |> int
                |> fun x ->
                    if x = swap_index1 then
                        if x > 0 then x - 1
                        else x + 1
                    else x

            genes |> Seq.mapi (fun i v ->
                if i = swap_index1 then genes.[swap_index2]
                else if i = swap_index2 then genes.[swap_index1]
                else v)
            |> Array.ofSeq

        //function that performs the crossing
        let produce_child parent1 parent2 =
            let cut_point = random() * float(Seq.length parent1 - cut_length
               ) |> int

            let parent1_part = parent1
```

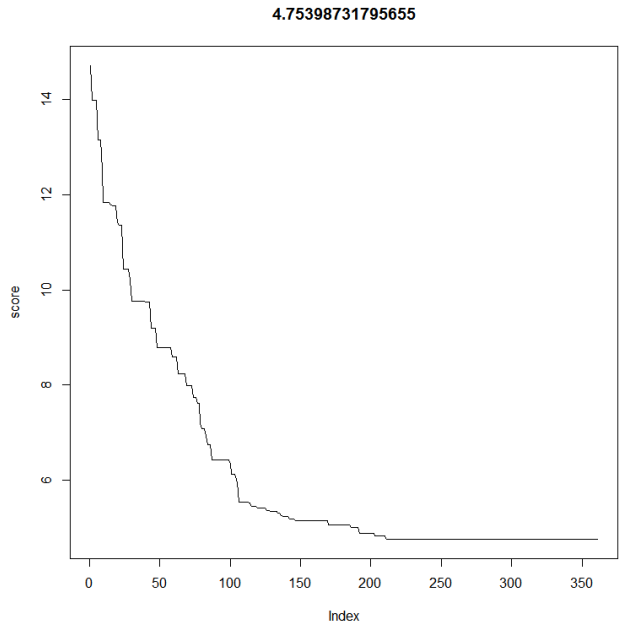
```
                            |> Seq.skip cut_point
                            |> Seq.take cut_length
            let parent2_part = parent2 |> not_in parent1_part

            let child =
                [parent2_part |> Seq.take cut_point
                 parent1_part
                 parent2_part |> Seq.skip cut_point]
                |> Seq.concat
                |> Array.ofSeq

            if random() < config.Mutationchance then
                mutate child
            else child

        let child1 = produce_child mother father
        let child2 = produce_child father mother

        child1,child2

    let rec loop scores flat_count (population: Position [] []) =
        match flat_count with
                //end case, the algorithm returns
        | count when count = config.FlatCount -> scores, population
        | _ ->  //loop case
            let mate_count = float(config.PopulationCount) * config.
               MatePercent |> int

            //performing crossovers and mutations on a selection of the '
               best'
            //population members (candidate solutions)
            let new_members =
                population
                |> Seq.take mate_count
                |> Seq.map (fun mother ->
                    let father_index =
                        float(config.PopulationCount) * 0.5 * random()
                        |> int
                    crossover mother population.[mate_count + father_index]
                    ||> fun c1 c2 -> [|c1; c2|])
                |> Seq.concat
                |> Array.ofSeq

            //adding the new population members to the population
            //discarding the 'worst' performing members
            //and sorting the members by performance
            let new_pop =
                [|new_members;
                  population
                  |> Seq.take (config.PopulationCount - 2*mate_count)
                  |> Array.ofSeq|]
                |> Array.concat
                |> Array.map (fun m -> Score m, m)
```

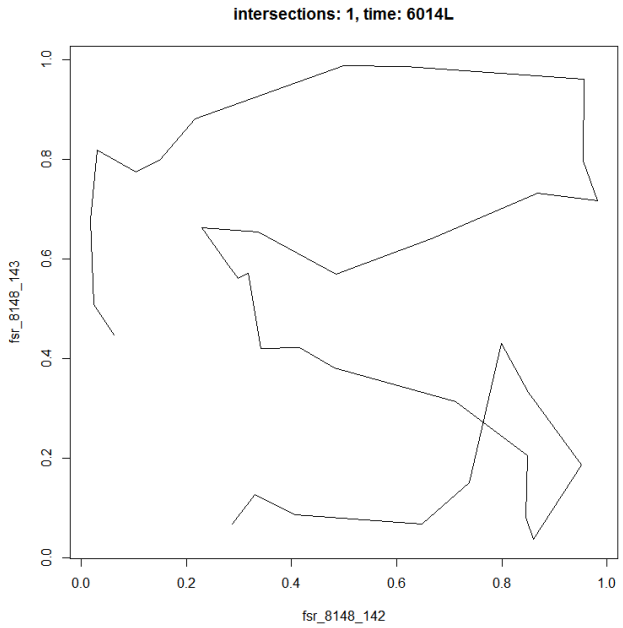
```
                |> Array.sortBy fst

            //compares current best score with previous best score
            //in order to determine when the members are not improving
               anymore
            //and the algorithm should return
            let score = fst new_pop.[0]
            let flat_count' =
                if Seq.head scores = score then
                    flat_count + 1
                else 0

            //invoking loop function recursively
            new_pop
            |> Array.map snd
            |> loop (score::scores) flat_count'

    //create an initial population and start the recursion
    let initial_pop =
        [|for _ in 1 ..config.PopulationCount  ->
            let genes_order = Shuffle genes
            let score = Score genes_order
            score, genes_order|]
        |> Array.sortBy fst
    let score = fst initial_pop.[0]

    initial_pop
    |> Array.map snd
    |> loop [score] 0
```

The algorithm is for example invoked with the following code

```
let stopwatch = System.Diagnostics.Stopwatch()
//generate list of positions of cities to visit
let cities = [|for _ in 1..35 -> random(),random()|]

//solve TSP with genetic algorithm
stopwatch.Start()
let genetic_config = {
    PopulationCount = 150
    FlatCount = 150
    Mutationchance = 0.5
    MatePercent = 0.25
    CrossOverPercent = 0.4
}
let scores2,solution2 =
    GeneticAlgorithm genetic_config cities
    ||> fun scores solution -> scores, Seq.head solution
stopwatch.Stop()

plot_solution (List.rev scores2) solution2 stopwatch.ElapsedMilliseconds (
   Seq.head scores2)
```

where the resulting plots are...





only one intersection makes this a fairly good solution

3.1 Simulated Annealing

Simulated annealing works by performing a 'random walk' where the strength of the randomness decays over time and where the candidate solutions are accepted with a probability scaling with with their performance. The benefit of performing a random walk and not accepting all solutions, is that it minimizes the risk of getting stuck in a local minima or maxima. Simulated Annealing can for example be used together with the gradient descend algorithm improve its chances on not getting stuck in local minima.

An implementation of simulated annealing for solving TSP can be found below. The proposed algorithm works by randomly swapping two indexes of cities in each iteration.

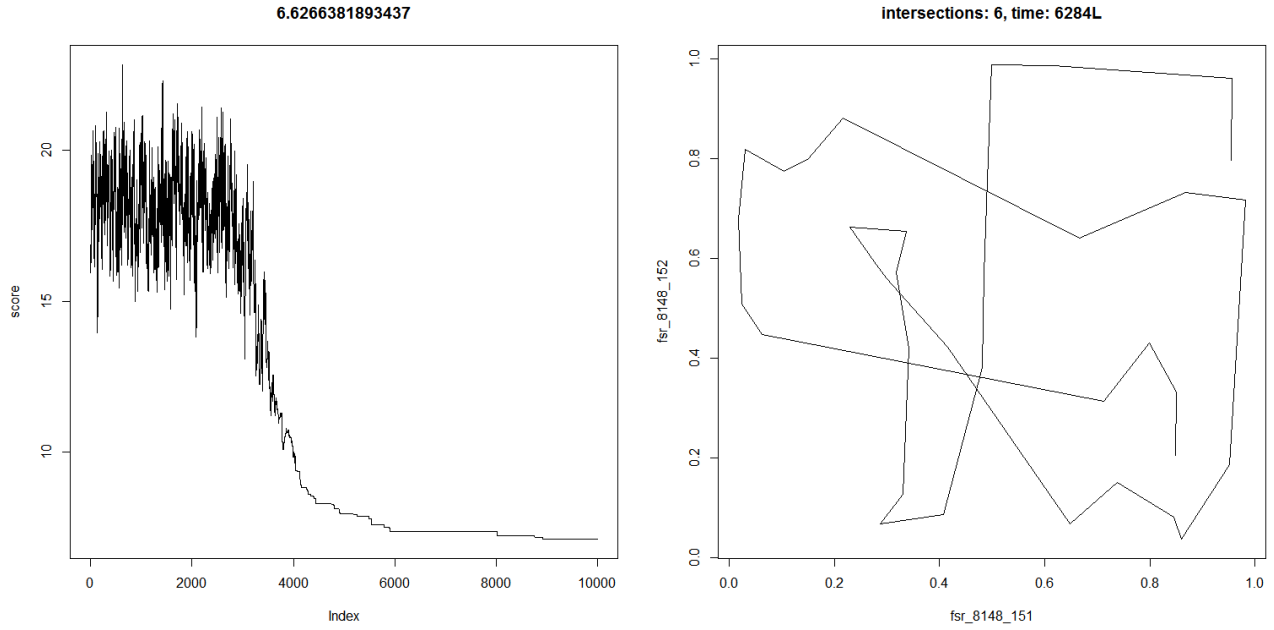
```
//function to swap two random indexes
let rswap_indexes (cities: Position []) =
    let swap_index1 = random() * (float cities.Length) |> int
    let swap_index2 =
        random() * (float cities.Length)
        |> int
        |> fun x ->
            if x = swap_index1 then
                if x > 0 then x - 1
                else x + 1
            else x

    cities |> Seq.mapi (fun i v ->
        if i = swap_index1 then cities.[swap_index2]
        else if i = swap_index2 then cities.[swap_index1]
        else v)
    |> Array.ofSeq

type Configuration = {
    StartTemp: float
    CoolingRate: float
}

//simulated annealing takes a configuration and a list of cities as
   arguments
//and attempts at finding an optimal permutation
let SimulatedAnnealing (config: Configuration) (cities: Position []) =
    let cities_count = Seq.length cities

    let rec loop prev_score cities temp = seq {
        let candidate_cities' = cities |> Array.copy |> rswap_indexes

        let score = Score candidate_cities'
        //calculating the probability of accepting the given solution
        let accept_prob =
            if score > prev_score then
                let delta_h = prev_score - score
                Math.Exp(delta_h / temp)
            else 1.0

        //decrease the temperature
        let temp' = temp * (1.0 - config.CoolingRate)

        if random() < accept_prob then
```

```
            yield score,candidate_cities'
            yield! loop score candidate_cities' temp'
        else
            yield prev_score,cities
            yield! loop prev_score cities temp'
    }

    let score = Score cities
    loop score cities config.StartTemp
```

The function is invoked like in the following code

```
let stopwatch = System.Diagnostics.Stopwatch()
//generate list of positions of cities to visit
let cities = [|for _ in 1..35 -> random(),random()|]

//solve TSP with simulated annealing
stopwatch.Start()
let solution' = cities
                |> SimulatedAnnealing {StartTemp=10000.0; CoolingRate=0.003}
                |> Seq.take 10000
let best_sol = solution'
               |> Seq.minBy fst
let scores = solution'
             |> Seq.map fst
             |> List.ofSeq
let solution = snd best_sol
stopwatch.Stop()

plot_solution scores solution stopwatch.ElapsedMilliseconds (fst best_sol)
```

With the resulting plots



when comparing this solution to the one generated by the genetic algorithm, it is clear that the solution produced by the genetic algorithm is a more optimal solution. However, simulated annealing converges much faster towards a good solution.

3.2 Using simulated annealing to fine tune the genetic algorithm parameters

As mentioned, simulated annealing can be used to fine tune parameters of other algorithms or models. Here I breifly discuss an approach to fine tuning parameters of the genetic algorithm.

In this approach, 4 parameters are modified by the simulated annealing algorithm to propose better solutions. The parameters are the population count, which determines the size of the population, mutation chance which defines the probability to mutate, mate percent whcih defines number of candidates to select for mutation and crossing, and cross percent which defines the number of 'genes' (or cities) to cross over from two population members (candidate solutions). It is assumed that population count is dependend on the number of 'genes', and it is therefore a scaling factor -and not the population count itself that the algorithm will attempt to optimize. Setting population count to a very high value, will make the algorithm find better solutions, but it will also slow it down.

First a scoring function is defined to evaluate the proposed configuration.

```
//scores a given configuration for a genetic algorithm
//the score is a ratio between line segment intersections and number of
   cities in relation to population count
let ScoreConfig (cities: Position []) (genetic_config: GeneticSolution.
   Configuration) =
    let score,solution =
        GeneticAlgorithm genetic_config cities
        ||> fun scores solution -> Seq.head scores, Seq.head solution

    let int_count = intersection_count solution |> float

    int_count / float(Seq.length cities) +  float(genetic_config.
       PopulationCount) / float(Seq.length cities)
```

Next the simulated annealing algorithm itself is defined. The algorithm doesn't differ much from the one displayed earlier for solving TSP. The most important things are the scoring funciton that uses the genetic algorithm to evaluate a given configuration and the adjustment of the parameters. The candidate parameters are calculated as normal distributions with the previous value as mean.

```
let SimulatedAnnealing candidate_conf candidate_score pop_count_ratio =
    let random_cities() =
        let cities_count =
            R.runif(1, 25, 50).AsNumeric()
            |> Seq.head |> Math.Floor |> int
        [|for _ in 1..cities_count -> random(),random()|]

    let rec loop prev_score n (pop_count_ratio: float) (conf:
       GeneticSolution.Configuration) = seq {
        let cities = random_cities()
        let temperature = 1.0 / Math.Log(float n)

        let pop_count_ratio' =
            R.rnorm(1, mean = pop_count_ratio, sd = 0.5).AsNumeric()
            |> Seq.head
            |> fun u ->
                if u < 1.0 then 1.0
                else if u > 4.0 then 4.0
                else u
        let mut_chance =
            R.rnorm(1, mean = conf.Mutationchance, sd = 0.1).AsNumeric()
            |> Seq.head
```

```
            |> fun u ->
                if u > 0.7 then 0.7
                else if u < 0.0 then 0.05
                else u
        let mate_percent =
            R.rnorm(1, mean = conf.MatePercent, sd = 0.1).AsNumeric()
            |> Seq.head
            |> fun u ->
                if u > 0.4 then 0.4
                else if u < 0.0 then 0.1
                else u
        let cross_percent =
            R.rnorm(1, mean = conf.CrossOverPercent, sd = 0.1).AsNumeric()
            |> Seq.head
            |> fun u ->
                if u > 0.4 then 0.4
                else if u < 0.0 then 0.1
                else u

        let candidate_conf = {
            PopulationCount = pop_count_ratio' * float(Seq.length cities) |>
                Math.Floor |> int
            FlatCount = conf.FlatCount
            Mutationchance = mut_chance
            MatePercent = mate_percent
            CrossOverPercent = cross_percent
        }

        printfn "candidate: %A" candidate_conf

        let candidate_score = ScoreConfig (random_cities()) candidate_conf

        let accept_prob =
            let delta_h = prev_score - candidate_score
            [1.0; Math.Exp(delta_h / temperature)]
            |> Seq.min

        let accept = R.rbinom(1, 1, accept_prob).AsNumeric() |> Seq.head

        if accept = 1.0 then
            yield (candidate_score, candidate_conf), pop_count_ratio'
            yield! loop candidate_score (n + 1) pop_count_ratio'
               candidate_conf
        else
            yield (prev_score, conf), pop_count_ratio
            yield! loop prev_score (n + 1) pop_count_ratio conf

    }

    loop candidate_score 2 pop_count_ratio candidate_conf
```

A solution is found with following code

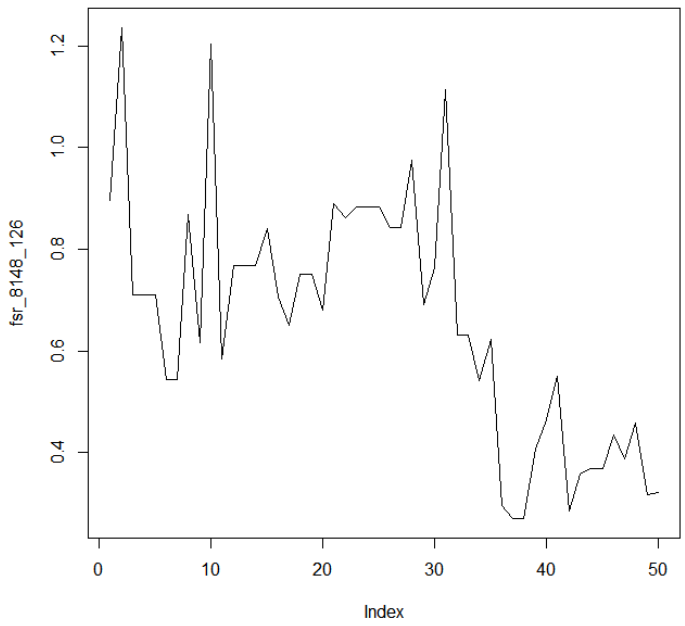
```
let cities = [|for _ in 1..35 -> random(),random()|]
```

```
let pop_count_ratio = 2.0

let candidate_conf = {
    PopulationCount = pop_count_ratio * float(Seq.length cities) |> Math.
       Floor |> int
    FlatCount = 50
    Mutationchance = 0.3
    MatePercent = 0.25
    CrossOverPercent = 0.3
}

let candidate_score = ScoreConfig cities candidate_conf

let results =
    SimulatedAnnealing candidate_conf candidate_score pop_count_ratio
    |> Seq.take 100
    |> List.ofSeq

let best = results
           |> Seq.minBy (fun x -> fst(fst x))
let best_conf = snd(fst best)
let best_pop_scaling_fac = snd(best)
```

which outputs following configuration

```
val best_conf : Configuration = {PopulationCount = 118;
                                 FlatCount = 50;
                                 Mutationchance = 0.4261533455;
                                 MatePercent = 0.2533570341;
                                 CrossOverPercent = 0.09119353916;}
val best_pop_scaling_fac : float = 2.700252446
```

which contains the configuration (in { .. }) and the proposed population scaling factor 3.86.



The error curve verifies the convergence as the algorithm progresses

The configuration for the genetic algorithm is tested with following code.

```
//test the proposed configuration
let cities2 = [|for _ in 1..100 -> random(),random()|]
let test = { best_conf with PopulationCount = int(50.0 *
   best_pop_scaling_fac) }
let scores2,solution2 =
    GeneticAlgorithm test cities2
    ||> fun scores solution -> scores, Seq.head solution
plot_solution (List.rev scores2) solution2 0 0.0
```

which produces following plots first with 50 cities and next with 100 cities...