MATH 3172 3.0
Combinatorial Optimization

Midterm I

Jacques Nel

February 18, 2020

1 Hill climb

1.1 Our implementation

Our naive implementation of the hill climbing algorithm is found in `hill_climb.py` in the function `hill_climb()`.

1.2 `grid1` and `grid2`

$$\texttt{grid1} = \begin{bmatrix} 3 & 7 & 2 & 8 \\ 5 & 2 & 9 & 1 \\ 5 & 3 & 3 & 1 \end{bmatrix}, \qquad \text{and} \qquad \texttt{grid2} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 8 & 10 \\ 0 & 2 & 4 & 8 & 16 \\ 1 & 4 & 8 & 16 & 32 \end{bmatrix}.$$

Code which sets up and calls `hill_climb()` for `grid1`, `grid2`, and `grid3` is found in the same file. The global maximum of `grid1` and `grid2` was trivial to find using a naive implementation of the hill-climb. Both adjacent and diagonal state transitions were allowed to reduce the number of iterations.

Table 1: Hill-climb for three given discrete functions

Function $f(x)$	iterations	time (μs)	$x^\star$	$f(x^\star)$	success
`grid1`	3	112	$(1, 2)$	9	yes
`grid2`	2	39	$(3, 4)$	32	yes
`grid3`	8	115	$(7, 98)$	-7.4	no

1.3 `grid3` is problematic

Observe in the table, in the previous section, the naive hill climbing algorithm fails to find the global maximum of `grid3` around the point $\boldsymbol{x}^\star = (1, 1)$.

Figure 1: 3D plot of $f_3(\boldsymbol{x})$

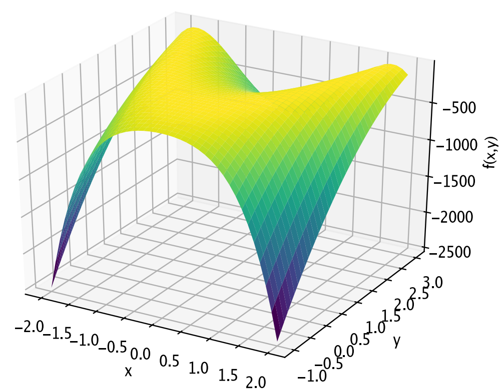
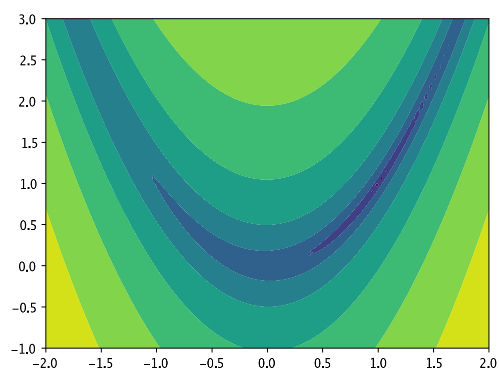


Figure 2: $(-f_3(\boldsymbol{x}))^{1/8}$ to emphasize narrow global maximum band



Especially when discretized, `grid3` has a ridge on which $\boldsymbol{x}^\star = (1, 1)$ lies. Hill climbing tends to get stuck along the sides this ridge, causing the algorithm to fail to find the global maximum. Aliasing, as a result of discretization, also results in several small isolated maxima near this ridge in the vicinity of $\boldsymbol{x}^\star$.

2 Simulated annealing

2.1 Our implementation

Our implementation of simulated annealing is found in `simulated_annealing.py` in the function `sa_solve()`.

2.2 `grid1` and `grid2`

The code which sets up and runs `sa_solve` for `grid1` and `grid2` is found is `sa_grid12.py`.

Table 2: Simulated annealing for `grid1` and `grid2`

Function $f(x)$	iterations	time (ms)	$x^\star$	$f(x^\star)$	success
`grid1`	21	1	$(1,2)$	9	yes
`grid2`	351	5	$(3,4)$	32	yes

For both `grid1` and `grid2` our implementation of simulated annealing finds the solution, although this technique is not ideal for these cases, which are well suited for hill climbing. Although we could tweak the cooling schedule and other parameters to achieve faster convergence, they are still about 1 order of magnitude slower than hill climbing.

Note: `sa_solve()`'s adpative setting had to be disabled for `grid2`.

2.3 `grid3`

Code which calls `sa_solve()` for `grid3` is found in `sa_grid3.py`.

2.3.1 Cooling schedule

An adaptive additive exponential cooling schedule[1] was used to find the global maximum of `grid3`.

$$T_k = T_n + (T_0 - T_n)\left(\frac{1}{1 + e^{\frac{2\ln(T_0 - T_n)}{n}\left(k - \frac{1}{2}n\right)}}\right) \tag{1}$$

Furthermore, we multiply T_k by an adaptive term $1 \leq \mu \leq 2$ which is calculated using the distance between the value of the current state $f(s_i)$ and $f^\star$, ie. the best value encountered so far[2].

$$T = \mu T_k = \left(1 + \frac{f(s_i) - f^\star}{f(s_i)}\right) T_k \tag{2}$$

[1] what-when-how.com, A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence)

[2] See footnote 1

In practice, we take the `np.abs` and use `np.clip(x, 1, 2)` to ensure that these assumptions are maintained.

The following parameters, initial temperature and number of cycles, denoted T_n and n respectivly, resulted in consistent convergence to the global maximum near $\boldsymbol{x}^\star = (0, 0)$:

$$T_n = 10 \qquad \text{and} \qquad n = 5000$$

2.3.2 Other cooling schedules considered

The following monotonic additive and multiplicative cooling schedules were also tried, but failed to produce good results:

1. Linear cooling,
2. (a) exponential multiplicative cooling,
   (b) logarithmic multiplicative cooling,
   (c) quadratic multiplicative cooling,
3. (a) linear additive cooling,
   (b) quadratic aditive cooling,
   (c) exponential additive cooling.

2.3.3 Results

With the above parameters, our implementation of adaptive simulated annealing consistently converged very close to $\boldsymbol{x}^\star$ after $k = 35754$ cycles, taking on average 1.31 seconds.

Due to the large number of steps k, instead of showing the whole state trajectory, the white line denotes an exponential moving average or EMA of the trajectory with $\gamma = 0.002$. Given the k^{th} state s_k, the EMA denoted Y_k is calculated recursively using

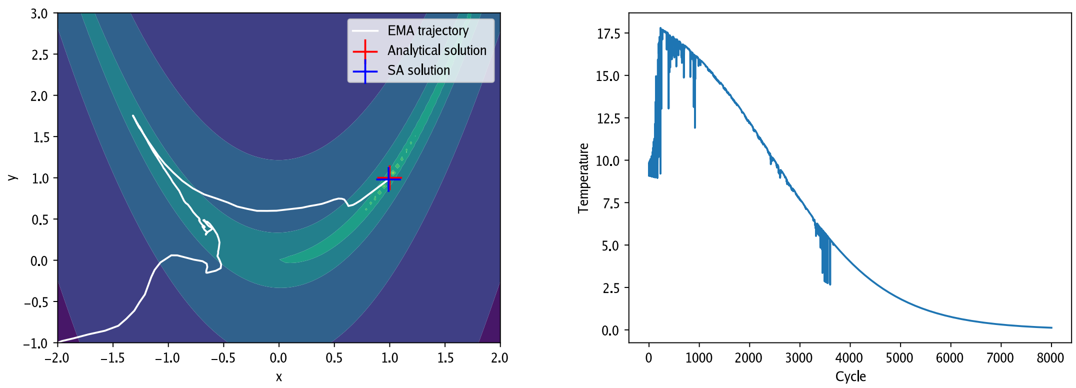
$$Y_k = \begin{cases} s_1 & \text{for } k = 0 \\ \gamma s_k + (1 - \gamma) s_{k-1} & \text{for } k > 1 \end{cases} \tag{3}$$

The blue cross denotes the solution found by our simulated annealing algorithm, and the red cross denotes $\boldsymbol{x}^\star = (0, 0)^T$ which is the analytical solution of $\boldsymbol{x}^\star = \text{argmax } f_3(\boldsymbol{x})$.

The figure on the right shows the exponential additive cooling schedule T_k being multiplied by a stochastic varrying term $1 \leq \mu \leq 2$.

Figure 3: Simulated annealing with adaptive exponential cooling schedule

(a) State trajectory (b) Adaptive-AE cooling schedule



2.3.4 Discussion

In case of `grid3` our implementation of simulated annealing is able to overcome the short-commings of the hill climbing algorithm in *section 1*. Given an appropriate cooling schedule, the state trajectory is able to 'jump' around in the isolated maxima contained inside the narrow ridge containing $\boldsymbol{x}^{\star}$. This is an execellent practical demonstration of simulated annealing's abilitiy to converge to to a global maximum, when other methods fail and/or get stuck in local maxima.

3 Traveling salesman problem

3.1 Our implementation

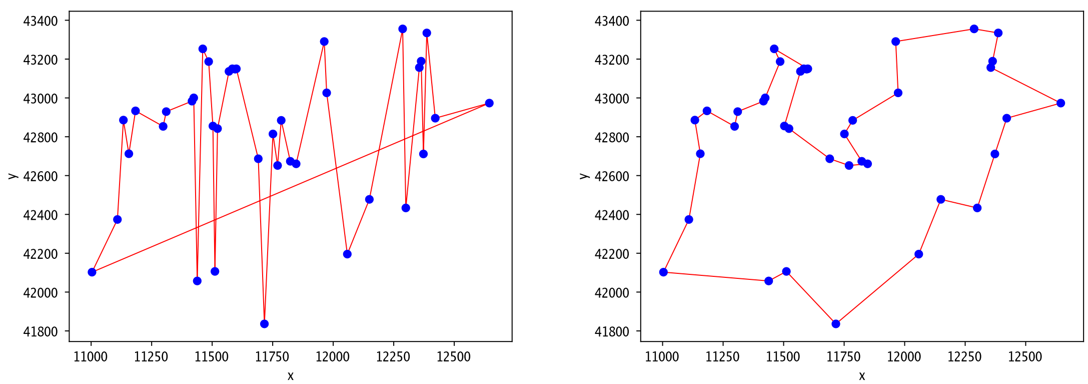
Our implementation of the `2-opt` TSP solver is found in `tsp_solver.py`. In addition to generating the figures below, the `dj38` problem is set up and solved in `dj38_solution.py`.

Note: The class `DJ38Loader` will download and parse the dataset from[3], so running this code requires internet connectivity.

3.2 Results

Figure 4: Using 2-opt to for `dj38`

(a) Initial tour (b) Optimum tour



Our implementation of `2-opt` finds the exact optimum tour after 20 iterations, in 318ms. The optimum tour length was found to be $f(\boldsymbol{t}^{\star}) = 6950$. Our results are identical with the results on the University of Waterloo's website[4].

[3] http://www.math.uwaterloo.ca/tsp/world/djtour.html DJ38 - Djibouti

[4] See footnote 3.