Heuristic Optimization Techniques
Exercise 2

Leon Schwarzäugl

November 22, 2023

1 Task 1

This exercise concerns the 0-1 Knapsack Problem: Assume a volume capacity $c_{vol} \in \mathbb{R}^+$ and set of goods represented by $G = \{1, \ldots, n\}$ with associated prizes $c_i, i \in G$, as well as volumes $v_i, i \in G$, are given. The goal of the problem is to find the selection of goods whose (i) summed volumes do not exceed c_{vol} and (ii) summed prizes are maximum.
Consider the 0-1 Knapsack Problem instance I given by $c_{vol} = 24, |G| = 7$, and

$$[(c_i, v_i i)]_{i=1}^{|G|} = [(22, 10), (4, 2), (8, 4), (18, 8), (16, 6), (8, 2), (18, 12)].$$

Start from the initial solution $x_0 := \{1, 2, 4\} \subseteq G$. Assume you have a random number generator yielding the values

$$[0.3, 0.05, 0.4, 0.1, \ldots, <\text{your own random values from } [0, 1) \text{ if needed}>]$$

in this order.
Perform by hand the Simulated Annealing (SA) algorithm as follows: Consider the list $L = [z_1, ..., z_\ell]$ of all feasible neighbors of the current solution sorted here in increasing order of their objective values; see below for the specification of the neighborhood.
Draw a random number ρ and assume $z_k = L[k]$ with $k = \lceil \rho \cdot \ell \rceil$ is the neighboring solution that is selected by the random neighbor step function within the SA. Decide if this solution will be accepted as new incumbent by the Metropolis criterion. Continue the execution of the SA algorithm until the second acceptance of a new solution.
The neighborhood $N(x)$ of a candidate solution $x \in G$ is given by all subsets $y \subseteq G$ where the symmetric set difference $x \ominus y = (x \cup y) \setminus (x \cap y)$ has exactly two elements and y is a feasible solution.
Assume the temperature within the SA to be $T = 2.86$ and ignore cooling in this example. What are the weaknesses of this specific SA approach? Explain what will be the problem if the instance size gets very large. How would you realize the selection of the next neighbor?

The given initial solution $x_0 = (1, 2, 4)_{44}$ (the subset number denotes the objective value, which is 44 for the initial solution) yields the neighborhood:

$$N(x_0) = \{(1)_{22}, (2)_4, (4)_{18}, (1, 2, 3)_{34}, (1, 2, 5)_{42}, (1, 2, 6)_{34}, (1, 2, 7)_{44}, (1, 4, 3)_{48}, (1, 4, 5)_{56}, (1, 4, 6)_{48}, (2, 3, 4)_{30}, (2, 4, 5)_{38}, (2, 4, 6)_{30}, (2, 4, 7)_{42}\}$$

This neighborhood consists of 14 entries, with the random number $\rho = 0, 3$ this yields $k = \lceil 0, 3 \cdot 14 \rceil = \lceil 4, 2 \rceil = 5$. The list L is now

$$L = [(2)_4, (4)_{18}, (1)_{22}, (2, 3, 4)_{30}, (2, 4, 6)_{30}, (1, 2, 3)_{34}, (1, 2, 6)_{34}, (2, 4, 5)_{38}, (1, 2, 5)_{42}, (2, 4, 7)_{42}, (1, 2, 7)_{44}, (1, 4, 3)_{48}, (1, 4, 6)_{48}, (1, 4, 5)_{56}]$$

which gives us $z_5 = L[5] = (2, 4, 6)_{30}$ (it's objective value is the same as $2, 3, 4)_{30}$, the ordering between these two was chosen randomly). Since $30 < 44$, this

solution has a lower objective value than the previous solution, which means that we will now be checking if this value will be taken nevertheless. We check the metropolis criterion:

$$e^{\frac{30-44}{2,86}} = 0,07 > 0,05 = \rho$$

Since with the random number $\rho = 0,05$ we fulfill the metropolis criterion, this solution is still accepted. T stays the same, since cooling is ignored in this example.
We calculate the neighborhood anew:

$$\begin{aligned} N((2,4,6)) = \{&(2)_4, (6)_8, (4)_{18}, (2,3,6)_{20}, (2,5,6)_{28}, (2,3,4)_{30}, (2,6,7)_{30}, (3,4,6)_{34}, \\ &(1,2,6)_{34}, (2,4,5)_{38}, (2,4,7)_{40}, (4,5,6)_{42}, (4,6,7)_{44}, \\ &(1,2,4)_{44}, (1,4,6)_{48}, (2,3,4,5,6)_{54}\} \end{aligned}$$

This neighborhood consists of 16 entries, with the random number $\rho = 0,4$ this yields $k = \lceil 0,4 \cdot 16 \rceil = \lceil 6,4 \rceil = 7$. The list L is now

$$\begin{aligned} L = [&(2)_4, (6)_8, (4)_{18}, (2,3,6)_{20}, (2,5,6)_{28}, (2,3,4)_{30}, (2,6,7)_{30}, (3,4,6)_{34}, \\ &(1,2,6)_{34}, (2,4,5)_{38}, (2,4,7)_{40}, (4,5,6)_{42}, (4,6,7)_{44}, \\ &(1,2,4)_{44}, (1,4,6)_{48}, (2,3,4,5,6)_{54}] \end{aligned}$$

which gives us $z_7 = L[7] = (2,6,7)_{30}$ it's objective value is the same as $2,3,4)_{30}$, the ordering between these two was chosen randomly). Since this solution is not better than the previous solution (in fact, they share the same objective value of 30), we again check the metropolis criterion with the next random number $\rho = 0,1$:

$$e^{\frac{30-30}{2,86}} = 1 > 0,1 = \rho$$

Since this result fulfills the metropolis criterion, we accept this solution, which marks the second acceptance of a new solution.

This SA approach has a weakness in the neighborhood structure chosen and in that we choose between different feasible solutions with an equal distribution of likelihood while also the occurrences of different in the feasible solutions differ to a high degree depending on their weight, regardless of their value contribution - this can be better pictured by considering for example the same neighborhood for another set of items

$$[(c_i, v_i i)]_{i=1}^{|G|} = [(1,4),(1,4),(1,4),(1,4),(1,4),(1,4),(400,40)], |G| = 40,$$

and the initial solution $(1,2,3)_3$ - because of the neighborhood structure the very efficient item 7 will not even be offered in the list of feasible solutions because of its great weight (it would need a symmetric set difference with 4 elements in order to be able to be found instantly) , it can only be found by first reducing to a solution of size 1. Even then, because of the uneven representation only one of the then feasible solutions will contain it (that is, the

solution $(7)_{400}$, which makes it quite unlikely to ever be chosen. Even in the task's example this effect is shown to a lesser degree by looking at $N((2,4,6)$ (the second iteration of the SA algorithm) - item 1, although more efficient than item 3 ($22/10 = 2,2 > 2 = 8/4$) is represented in 3 of the 16 entries, whereas item 3 is represented in 4 of them, with both of them not being in the solution at the start of that iteration step. Of course a more efficient item should not always be in the optimal solution if the capacity is not fully used, as shown in the optimal solution $(1,2,3,5,6)_{58}$ with summed capacity $\sum v_i = 24$ as opposed to the solution $(2,3,4,5,6)_{56}, \sum v_i = 22$, where the more efficient item 4 is chosen instead of the less efficient item 1, but making worse use of the given capacity. Still that contributes to the problem.

These findings suggest that for (much) larger instance sizes, we should not choose the candidate using a uniform chooser, but instead for example with a weighting system depending on the quality of the solution. Also the neighborhood structure could be modified to allow symmetric set differences with 1 or 2 elements, for example.

2 Task 2

For the MaxSAT problem on an instance made up of variables $v_i, i = 1, \ldots, n$, and clauses $C_i, i = 1, \ldots, m$, consider the following selection strategy of a neighbor in the 1-flip neighborhood:
"Determine the number g_i of additionally satisfied clauses (with respect to the current count of satisfied clauses) when variable $v_i, i = 1, \ldots, n$, is flipped and select $i^* \in g_i : i = 1, \ldots, n$ (in case the choice of i^* is not unique, the lexicographically prior variable is selected)"

(a) Find a small MaxSAT instance $\mathcal{A}$ on variables $(v1, \ldots, v_n)$ simultaneously fulfilling the following properties:

- Starting from the truth value assignment $(v_1, \ldots, v_n) \mapsto (0, \ldots, 0)$, selecting the 1-flip neighbor y_{next} according to the above strategy (i.e., y_{next} will have variable v_i flipped) and visiting y_{next} as successive solution, never will reach a global maximizer of $\mathcal{A}$, even if this process would be re-applied arbitrarily often to its last visited solution.
- A Tabu Search (add a brief description) relying on the 1-flip neighborhood structure and ranking non-tabu 1-flip neighbors according to their above defined gain g_i visits the global optimum of the problem instance after a certain number of steps.

(b) Find a small MaxSAT instance $\mathcal{B}$ on variables $(v1, \ldots, v_n)$ simultaneously fulfilling the following properties:

- Starting from the truth value assignment $(v_1, \ldots, v_n) \mapsto (0, \ldots, 0)$, selecting the 1-flip neighbor y_{next} according to the above strategy (i.e., y_{next} will have variable v_i flipped) and visiting y_{next} as successive solution, never will reach a global maximizer of $\mathcal{B}$, even if this process would be re-applied arbitrarily often to its last visited solution.
- A Configuration Checking (CC) algorithm (add a brief description) relying on the 1- flip neighborhood structure and ranking 1-flip neighbors (whose configuration indeed changed in the meantime) according to their above defined gain g_i visits the global optimum of the problem instance after a certain number of steps.

```
procedure tabu search

begin
  TL ← ∅; // tabu list of length t_L
  x ← initial solution;
  repeat:
    X' ← subset of N(x) regarding TL;
    x' ← best solution in X';
    add x' to TL;
    remove element from TL, which is older than t_L iterations;
    x ← x';
    if x is best solution found so far then
      store x as best solution;
  until stopping criteria satisfied;
end
```

Figure 1: Tabu Search

2.1 (a)

Such an instance can be found by having the first variable (in fact, this could be any variable, but then the wording on the following would have to be a bit different, because if two flips would not change the number of additionally satisfied clauses, then the first one will be lexicographically chosen) having no bearing on the clauses while starting from a state in which every step in the 1-flip neighborhood leads either to the number of additionally satisfied clauses being 0 or negative. Such a configuration is given by $(v_1, v_2, v_3) \mapsto (0, 0, 0)$ and the clauses

$$(\neg v_2 \vee v_3) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_3 \vee v_2) \wedge (\neg v_3 \vee v_2) \wedge (v_2) \wedge (v_3)$$

The initial state fulfills 4 clauses. The 1-flip neighborhood consists of (again the subset number represents the number of additionally satisfied clauses):

$$N((0,0,0)) = \{(1,0,0)_0, (0,1,0)_{-1}, (0,0,1)_{-1}\}$$

Of these, $(1, 0, 0)$ is the best solution and is chosen. Now, the neighborhood looks like

$$N((1,0,0)) = \{(0,0,0)_0, (0,1,0)_{-1}, (0,0,1)_{-1}\}$$

Of these, $(0, 0, 0)$ is the best solution and is chosen.

We can see that for both states, the 1-flip neighborhood cannot find an improvement and loops endlessly, flipping a variable that does not change the outcome, while never finding either of the best solutions $(0, 1, 1)$ or $(1, 1, 1)$.

The idea of tabu search is to implement a list of solutions that have already been visited, and forbidding for them to be visited again until they have reached a certain "age". With this, the above cyclic behavior can be avoided:

1. In the first step, we again choose $(1,0,0)$ as in the above, with the difference being that we now add $(1,0,0)$ to the tabu list (let t_L, the number of iterations after which to eliminate an element from the tabu list, be big here).

2. In the next step we again backtrack to $(0,0,0)$, also adding it to the tabu list, which now is $\mathrm{TL} = [(1,0,0),(0,0,0)]$.

3. In the next step, $(1,0,0)$ is disregarded as it is on the tabu list, which leaves $(0,1,0)_{-1}$ and $(0,0,1)_{-1}$, of which $(0,1,0)$ is chosen lexicographically, and we add $(0,1,0)$ to the tabu list, now containing $\mathrm{TL} = [(1,0,0),(0,0,0),(0,1,0)]$.

4. For the last step, we check (with currently 3 clauses being fulfilled) the allowed subset of the neighborhood, which is $\{(1,1,0)_0,(0,1,1)_3\}$, from which we choose $(0,1,1)$, at which point we have found one of the optimal solutions.

2.2 (b)

```
procedure Local Search for SAT ({x_1, ..., x_n}, {C_1, ..., C_m})

begin
  s ← initial truth value assignment for x_i, i = 1, ..., n;
  Initialize array of Booleans configCh_{x_i} ← 1, i = 1, ..., n;
  repeat:
    if s is a satisfying assignment then return s end
    Pick variable z ∈ {x_1, ..., x_n}; // by a heuristic
    if configCh_{x_i} = 1 then
      s ← s with z flipped;
      configCh_z ← 0
      for y ∈ N(z) do configCh_y ← 1 end
    end
  until stopping criteria satisfied;
  return no solution found
end
```

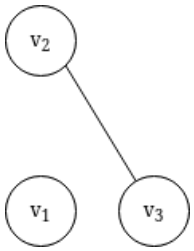

Figure 2: Configuration checking algorithm and visualization of neighborhood structure for the given set of clauses $(\neg v_2 \vee v_3) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_3 \vee v_2) \wedge (\neg v_3 \vee v_2) \wedge (v_2) \wedge (v_3)$

For configuration checking, we again follow the same goal of avoiding cyclic behavior; this time (for our practical approach), we save apart from the solution vector a vector of "bits that are currently allowed to be changed", initialized to all 1's - whenever we flip a bit, we set that bit's corresponding entry in that array from 1 to 0, then set all it's neighbors corresponding bits in that array to 1.

For the task, in fact the same set of clauses used for (a) also fulfills this assignment, however we have to apply a tiny change to the configuration

checking algorithm shown: when choosing the variable to be flipped, it should only be sourced from variables whose $\text{configCh}_{x_i} = 1$, as otherwise with our chosen heuristic we will only ever look at the same element (as stated in the task description as well as above, I just wanted to point out this difference in regards to the algorithm from the image).

In detail: We start from $(0, 0, 0)$, with $\text{configCh}_{v_i} = (1, 1, 1)$. Our selection strategy decides to check if we should flip v_1 (with the number of additionally satisfied clauses being 0), which would result in $(1, 0, 0)$/ $\text{configCh}_{v_1} = 1$, so we flip v_1, receive $(1, 0, 0)$, and set $\text{configCh}_{v_1} = 0$. From the in the meantime changed configurations we now check v_2 (with the number of additionally satisfied clauses being -1, chosen lexicographically). $\text{configCh}_{v_2} = 1$, so we receive $(1, 1, 0)$, set $\text{configCh}_{v_2} = 0$. We do however not change configCh_{v_1} back to 1, as it is not a neighbor of v_2. Lastly, we select v_3 (with the number of additionally satisfied clauses of 3), see that its $\text{configCh}_{v_3} = 1$, and receive one of the optimum solutions $(1, 1, 1)$ (while also settings $\text{configCh}_{v_3} = 0$ and $\text{configCh}_{v_2} = 1$.

3 Task 3

Design a Genetic Algorithm that solves Sudokus.

(a) Describe what a Chromosome looks like.

(b) How will the fitness function look like? Is it possible that you produce invalid solutions? If so how do you plan to handle those?

(c) Discuss how to apply, at least two different recombination operators to your solution. Design at least one of the recombination operators yourself specifically to be applied to Sudoku. This operator should utilize the rules of Sudoku. Discuss how you expect them to perform.

(d) Design two mutation operators and describe how to apply them to your solutions. What performance do you expect of these operators?

(e) Can your GA design be expanded to solve larger Sudokus containing the numbers 1 through n^2 where $n \in N$ on a n^2 rows, columns and boxes. If so how can you expand your design

For this algorithm, we can use a chromosome that consists of a list of all the numbers that have been filled in the sudoku. To find out to which row an entry belongs, we take that entries index integer divided by 9 (so for example the entry with index 9th belong to the row with index 1 - so the second row), to get the corresponding column we take the modulo of 9 of the entry index (so the same index 9 would yield the column with index 0

- the first column). The initial filling of the sudoku with values fills each of the nine boxes with values from 1 to 9, taking into account the initial values. That way, we at least know that the values within the boxes are "correct" as in the way that they do not instantly violate a rule of sudoku.

Since we realistically need to expect that we will never solve a sudoku with a genetic algorithm, the fitness function should apply a penalty term for each entry that is invalid. An invalid entry is here defined as an entry that has a number $[1, 9]$ that is also found in another place in the same row or column - so a row that is filled with $[1, 2, 3, 4, 5, 6, 7, 8, 8]$, will apply one penalty (not two, since one of the instances of 8 does not violate the sudoku rules). Again, realistically we will never find the true solution, so we try to find solutions with a small penalty term.

One recombination operator that can be applied is "uniform box crossover", a uniform crossover for boxes - we decide for each box from which parent it takes the box. When we apply any operator, we need to make sure not to break the initial values of the sudoku. Luckily, that is automatically given, since for two solutions those will always respect the initial values.
Another operator that can be used is "1-point box crossover", where we perform a 1-point crossover on box scope (thinking on index level, at the index that is at the start or end of a box). Again, no sudoku rules will be violated inside boxes since the parents will have the initial values correctly set each time. For both of these, I expect bad performance, simply because sudoku is a game where each value is extremely dependant on all others, which is why we cannot expect to gaina lot from such an approach. That being said, the 1-point crossover should perform better, since it at least retains some of the relation between boxes, while the uniform crossover leads to a de-facto randomization of boxes.

For mutation operators we need to take special care of these initial values. The mutation operator that is the most fundamental for this scenario is the "swap" mutation, which simply swaps two values within a box. For this operation, it is forbidden to target initial values.
Another mutation operator would be permutation within the box - all non-initial values within a box are incremented by 1 and then modulo 9 is applied. Of course, theoretically this mutation can be reduced to a sequence of swap-mutations. This time, I expect both of these operators to perform equally bad, since neither of them retains any relation to boxes outside the scope.

If nothing else, this approach can be easily expanded to "solve" larger sudokus - simply expand the number of boxes, the numbers contained within, or anything else, it does not really matter.

4 Task 4

Design a variant of ACS that solves Sudokus.

(a) How does the construction graph look like? How many edges and nodes does the graph have?

(b) How does an ant set a solution and what objective function can be used?

(c) How does the pheromone update work? What initial values for the pheromone matrix seem to be reasonable?

(d) What information can we use during construction to prune the construction graph?

__TODO__ CONSTRUCTION GRAPH IMAGE

For a standard sudoku of 9×9 size, the construction graph consists of a "location"- as well as a "value" domain, with each ant travelling between these. A travel from a location to a value has the meaning of setting a value in a specific spot in the grid, a travel from a value to a location has the meaning of the time when the ant visited that spot - this is important since an early move is a lot less unconstrained than a move later on - for example the last move has no degree of freedom, and as such should not be weighed too highly when considering its impact on the solution quality. As such we receive $81 \cdot 2 = 162$ nodes in the construction graph, as well as $(2 \cdot 81) - n_{init} \cdot 2 - 1$ edges, where n_{init} is the number of initial values given. The last -1 is derived from the information that the ant does not need to perform a round-trip; when it reaches the last node in the "value" domain, it's path is completed. This information is also what we can use to prune the construction graph, as we do not need to travel through initial-value locations, as their value is already fixed.

An ant sets a solution by first choosing a location to be visited; for each location, it chooses a value $[1, 9]$ that has not yet been set in the respective box, and then travels to another location, continuing until the grid has been filled. As an objective function the same penalty based approach described in task 3 seems reasonable. An ant that has produced a small penalty term will leave more pheromones than an ant that has produced a high penalty term.
Initial values for the pheromone matrix should be as such so that every node is equally likely to be visited (of course, again, nodes with initial values do not need be visited at all). For the pheromone update, more pheromone should be added the sooner a move is done by an ant, as, as described above, such an earlier move has more freedom and is as such more indicative if the chosen path has been of good quality.