NatAlgReport

Name: Theo Farrell User-name: nchw73

Two-letter code for your chosen NatAlgReal algorithm: CS

The best minimum I achieved was at [7.922588886680929, 7.942046903938513, 7.930403753699924, 7.926600109343327] with a value of -62.15358247837975. This took 9.9 seconds. The parameters were num_cyc = 24000, N = 50, p = 0.6, q = 0.25, alpha = 1.6, beta = 1.5.

Detail any minimum (or minima) that you have found, along with the parameter settings and the elapsed time. I am looking for a performance-related account of how your implementation behaved under various parameter settings, along with any insight you might have gained as to the general efficacy of your chosen algorithm under various conditions.

Two-letter code for your chosen NatAlgDiscrete algorithm: CS

I chose the graph partition (GP) problem, where each nest represents a possible partition of the vertices.
My best values for each graph were: A – 159, B – 268, C – 212.

Discretization - To discretize the algorithm, I created discrete Levy and local flight functions. To accurately represent the philosophy behind cuckoo search, I ensured that local flights from a nest only minimally disrupted its associated partition, and that Levy flights from a nest would generally be small changes with the chance of a large disruption to the existing partition. In this way, I defined a local flight as randomly swapping two vertices in different partitions, and (initially) a Levy flight as doing this M times, where M is the integer part of *alpha * Levy_step*, where the step is sampled form a Levy distribution and alpha is the Levy scaling factor. Without any enhancements, this gave positive results. Within a minute, I halved the initial number of conflicts in graphs A (298 -> 173) and B (626 -> 382), and reduced graph C conflicts by about a third (779 -> 509). This was impressive since the final number of conflicts for each graph after 5000 cycles was only ~10% fewer than the smallest number of conflicts found in 1 minute. This suggests the discretization works, since the convergence was fast as expected in cuckoo search. However, this speed is also partly due to my method of calculating conflicts which avoided checking the entire adjacency matrix and used a custom *Vertex* class to store neighbours instead (explained in *get_conflicts* function in code).

(continued over)

Parameter tuning – Number of nests (**N**) had a significant impact on runtime and sometimes convergence time. In graph A, for example, during one test when N = 50, after 1 minute the minimum number of conflicts found so far was 181. When N = 20, then it was 171. This highlights the exploration vs. exploitation trade-off in cuckoo search, where larger N encourages greater exploration of the search space and smaller N encourages exploitation of the best nests found so far. However, larger N significantly affected run time for larger graphs, so I decided on a higher N for graphs C and B than A.

Adjusting **p** (fraction of nests to locally search) and **q** (fraction to abandon) was also crucial to this trade-off. A low p and high q resulted in very fast convergence, whereas high p and low q caused the opposite but allowed greater exploration. If q = 0, the convergence was especially slow (conflicts > 200 after 1 minute), since the only way to get a better solution was to find it via flights. I found that for larger graphs, a larger p and q was more useful. In graph A, it generally converged to ~170 for any p in [0.4, 0.9] and any q in [0.1, 0.9] because the graph (and search space) was quite small, and local flights are fast operations. However, for graph C, when p = 0.8 and q = 0.4 it reached 470 conflicts after 30 seconds. For p = 0.5 and q = 0.25 it reached 480 conflicts. After longer than 30 seconds, searches with larger p and q converged substantially quicker than those with smaller p, q.

Enhancements

- **Alpha decay** and **Levy flights from new nests** (https://doi.org/10.1016/j.chaos.2011.06.004) – When generating a new nest to replace an abandoned one, I do a Levy flight from it after generation. I introduce a new parameter, alpha$_t$, which decreases as t increases and is initialised to alpha. Alpha$_t$ is only used for these Levy flights. This encouraged exploration when generating new nests, but the decaying alpha meant that these Levy flights would eventually be ineffective, improving convergence speed. This was effective when combined with the ranked nest strategy below.
- **Ranked nest generation** – Inspired by ranked ACO, I select w (user-chosen parameter) of the best nests found in a given cycle. When replacing abandoned nests, the newly generated nest is a Levy flight from one of these best nests, selected with probabilities proportional to their fitness. I also added a *p_ranked* parameter, which is the probability that a newly generated nest will be one I have just described, as opposed to a typical new random nest. I liked *p_ranked* = 0.8 and w = 6 for good exploitation of the best nests found so far, and because allowing 20% of newly generated nests to still be completely random could help escape local optima. This vastly improved convergence time. In graph C for example, solutions fewer than 300 conflicts were consistently found within 1 minute.
- **Fiduccia–Mattheyses (FM) -inspired Levy flight** – Inspired by the FM heuristic, this Levy flight first calculates M as usual. Then 2 random vertices are selected and swapped. Their neighbours are added to a checklist. Since these neighbours might be best placed with the vertex that has just been swapped to a different partition, all these neighbours are checked and swapped as well if the resulting number of conflicts is reduced. In total, a maximum of M swaps occur. This could be quite time consuming, hence I added a *neighbour_limit* parameter which limits the number of neighbours of each vertex considered in this search. Unfortunately, this Levy flight operation didn't have a significant impact on the results compared to the vanilla operator.
- **Greedy nest generation** – I attempted to seed the initial population of nests with some 'greedy' partitions. These were created by taking the vertices in a random order and building the partitions up one vertex at a time. Each vertex is added to the smallest partition that doesn't conflict. If that's not possible, it is added to the partition with the smallest number of conflicts. Unfortunately, this actually made the initial population seemingly worse. While a completely random initial population generally had an initial minimum of conflicts = 780, a 'greedy' initial population tended to start with an initial best of 880! They gradually converged to the same number after several cycles, but this enhancement was not useful in improving results.