Armen Ourfalian
May 23, 2017

Comp 610 Project

I spent quite a bit of time deciding what data structures I wanted to use for this project. Initially, I used the following:

- An ArrayList of size n that held the clauses
    - Each clause was an int[2] that held the literals
    - The ArrayList was originally a LinkedList, which severely hindered performance (mostly because of the way I iterated through the list)
- The literals were kept in a single BitSet object that kept track of the current values of each literal.
    - Any time I wanted to check an answer, I would clone the BitSet, replace its data, check the solution, and restore the original BitSet from the clone
    - The size of the BitSet was set equal to the highest indexed literal in the problem
        - This approach saw problems right away with test1, as it had no mechanism for ignoring the literals that were never used

At this point, I decided to reorganize the way I stored the problem instance, and I also decided to separate the storing of the instance from the solving of it(the Solver class solves the instance, while the class Max2SAT loads the instance from the input file)
The new data structures work in the following way:

- An ArrayList of size n that holds the clases
    - The clauses are now their own class: Clause
        - An instance of the class Clause holds two Literal objects, and is responsible for checking its own satisfiability
        - A clause also checks itself (before it wrecks itself) to see if it's a tautology. If so, a lot of its initialization and sat-checking is avoided because the clause will always be true
- A HashMap of size (Max index of all the literals)
    - Note that although the size of the HashMap is just as big as the size of the BitSet above, the HashMap does not get fully populated
    - The literals are also their own class: Literal
        - This class mostly does bookkeeping on itself, keeping track of how many times it appears in the problem (also how many times it appears as a negative)
        - I was planning on making better use of this metadata, but it didn't happen, so now it just gets printed into an output file
    - The HashMap is mostly used to initialize the problem, after the Solver takes over, the HashMap is accessed through various BitSet objects.
- Multiple BitSets to check various solutions to the problem
    - I made use of some helper functions that mapped a BitSet to the HashMap and vice versa. These were very useful.

Armen Ourfalian
May 23, 2017

Comp 610 Project

This program reads an instance of MAX2SAT and attempts to find a reasonable solution by doing repeated local searches. There are two local search algorithms I implemented, labelled localSearchMethodical () and localSearchJumpy ():

- localSearchMethodical:
  - Defines a solution as a set of 0's and 1's (BitSet) that represents the truth assignment of the literals.
  - A neighborhood is any solution that is different by x bits, where x is an input parameter.
    - If $x \leq 1$, then a neighborhood is any solution that is different by a single bit.
    - If $x > 1$, then this function recursively calls itself x times, creating a wider neighborhood
      - For that reason, the input parameter is labelled width (not x)
  - The move condition is defined as: check the entire neighborhood and move to the highest-valued neighbor
  - The stop condition is defined as: stop when you can't find any better neighbors
- localSearchJumpy:
  - Defines a solution as a set of 0's and 1's (BitSet) that represents the truth assignment of the literals (same as above)
  - A neighborhood is any solution that is different by a single bit (same as the above case $x \leq 1$)
  - The move condition is defined as: move to the first neighbor that is higher-value (hence the name jumpy)
  - The stop condition is defined as: stop when you can't find any better neighbors (same as above).

I quickly found that for most of the test cases (the latter 80% to be exact), any neighborhood larger than 1 was unfeasible due to the large number of literals present (otherwise localSearchMethodical algorithm, with a width of 3 was often returning the optimal answer, but at the cost of my youth and sanity). This was actually the reason I implemented localSearchJumpy in the first place.

In class, the Metropolis Algorithm really interested me, and I wanted to use a similar approach for this project. So experimented a little with an algorithm that randomly does one of three (or four) things: get a random solution (fastest), check the immediate neighborhood of the current solution, do a localSearchMethodical(width =1), do a localSearchMethodical(width > 1) (obviously the slowest). The idea here was to do the really slow procedures the fewest number of times.

I spent a lot of time configuring this algorithm: tweaking the probabilities, changing the actual procedures, etc. but did not see any significant improvements in the results, which fluctuated widely. Furthermore, I found that any time the slow procedures' probability was high enough that they were executed more than once (by random chance), they slowed down the performance too much.

Armen Ourfalian
May 23, 2017

Comp 610 Project

The final solution (oops, didn't mean to quote Hitler) I settled on utilizes a bit of the randomness of my Metropolis-like algorithm, without relying on finely-tuned probability ratios. I called the algorithm compoundLocalSearch, because it does local searches on top of local searches .

1. This algorithm begins with a random BitSet, runs the localSearchJumpy algorithm on it (which is faster than localSearchMethodical)
2. If the resulting solution is the best one yet, then save it.
3. If it's not, then reduce the number of tries remaining (think: extra lives in a video game).
4. After no more tries are left (game over), take the best possible solution found, and run localSearchMethodical(width = 2) on it.
    a. Although this last step takes up a lot of time, it only happens once during the entire algorithm
    b. Furthermore, since the value of the solution at this point should be quite high, the algorithm should run more quickly

Finally, almost being satisfied, I wrapped the above steps 1 – 4 inside of a loop to repeat a few times (it is important that this number be small compared with the number of tries) and return the best answer among all the local searches performed.