**All code is available at:** asultan123/FLP_Scheduling (github.com)

**The first algorithm will be some version of a steepest descent local search algorithm. Develop a set of algorithms that find good initial solutions without spending too much time. You should use at least two such algorithms, and they can be based on any of the approaches you have worked on so far (greedy, exhaustive, branch and bound, random, etc.)**

Initial solutions for our steepest descent local search algorithm are found with either a random or a greedy approach, both of which resolve the initial solution very fast. Random solutions are found simply by assigning each task (node) to a random processor. Greedy solutions are found based on previously explained processor-task greedy binding applied to the grouped topological sort of the particular task precedence graph being solved. These algorithms give an initial binding where each task is assigned to a particular processor, and based on this binding the schedule function (the same used in the exhaustive and greedy algorithms) assigns a start time to each task. The makespan (defined as the maximum assigned start time, plus one) of the solution is the magnitude used as the 'cost' value of the steepest descent local search algorithm.

**Define the neighborhood of each solution and use the local search algorithm to iteratively search for a local minimum.**

We define the neighborhood as a change in the binding of a task from one processor to another. Therefore, the total neighborhood size is node_count*(processor_count - 1). This intuitively looks like a good neighborhood since it covers the points that are closest to our current solution in every possible 'direction', and it is small enough to be covered in reasonable time.

The steepest local search algorithm computes the makespan of each member of the current solution, and selects the smallest one. If this best member of the neighborhood is smallest or equal to the current solution, it is selected as the new current solution (we also select an equal neighbor, to allow for a wider exploration of the neighborhood). This process is repeated until no better (strictly smaller, not equal) neighbor is found after a predefined number of neighborhood explorations (in our simulations we used 20).

**Evaluate a second algorithm selected from the following list: tabu search, branch and bound, dynamic programming, variable-depth search, simulated annealing, or genetic algorithms.**

We evaluated a Genetic Algorithm as our second local search algorithm. The specific variant of the GA algorithm used is the Breeder algorithm defined in [1]. The steps of the algorithm are:

0. Define a genetic representation of the problem.

In our example each possible binding for a schedule is an individual in a population. The genes represent the individual task processor bindings.

1. Create an initial population P(0) of size N. Set t = 0.

Populations are created randomly.

2. Each individual may perform local hill climbing.

The variant of local hill climbing used here is one iteration of the steepest descent algorithm described above.

3. The breeder selects T% of the population for mating. This gives the selected parents.

The selection is based on  sorting members of the population based on fitness and selecting a predefined number of them called the cut-off. Fitness for each individual is calculated as the ratio between the total average makespan of the population and the makespan of the individual binding post scheduling.

4. Pair the parents at random forming N pairs. Apply the genetic operators crossover and mutation, forming a new population P(t + 1). 5. Set t = t + 1 and return to Step 2.

Mutations is applied as follows, for each individual there is a uniform probability of mutation called the mutation rate. If mutation is to occur a random task's binding is changed to a random target processor.

Crossover is applied as follows, for each random pair of parents the resulting offspring takes bindings from each parent such that there is a equal probability of choosing a binding from either parent.

The termination condition for the above algorithm is no change in overall population fitness for a predetermined number of steps. The tolerance for what constitutes "no-change" is an tunable parameter.

The parameters used for the benchmark run in Fig. 1 are:

population_size = 50

cut_off = 25

mutation_rate = 0.25

fitness_tolerance = 0.25

max_steps_with_no_change = 20

**Make sure to include data describing run times as a function of instance size, and comparisons between all solutions you have found for each instance, including the greedy, ILP, and exhaustive algorithms, and the LP lower bound.**

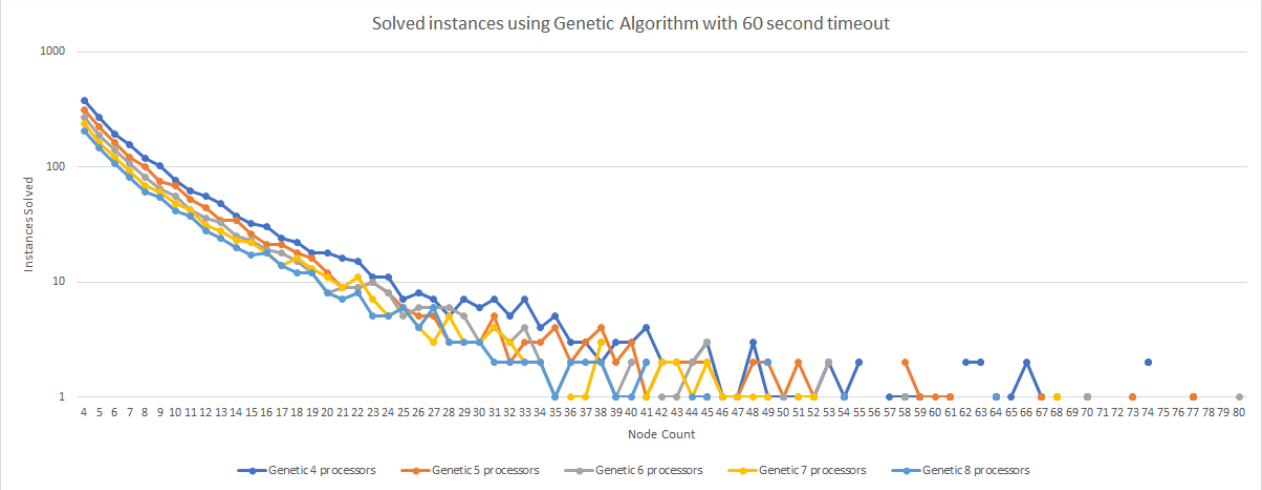


Fig. 1 GA benchmark performance

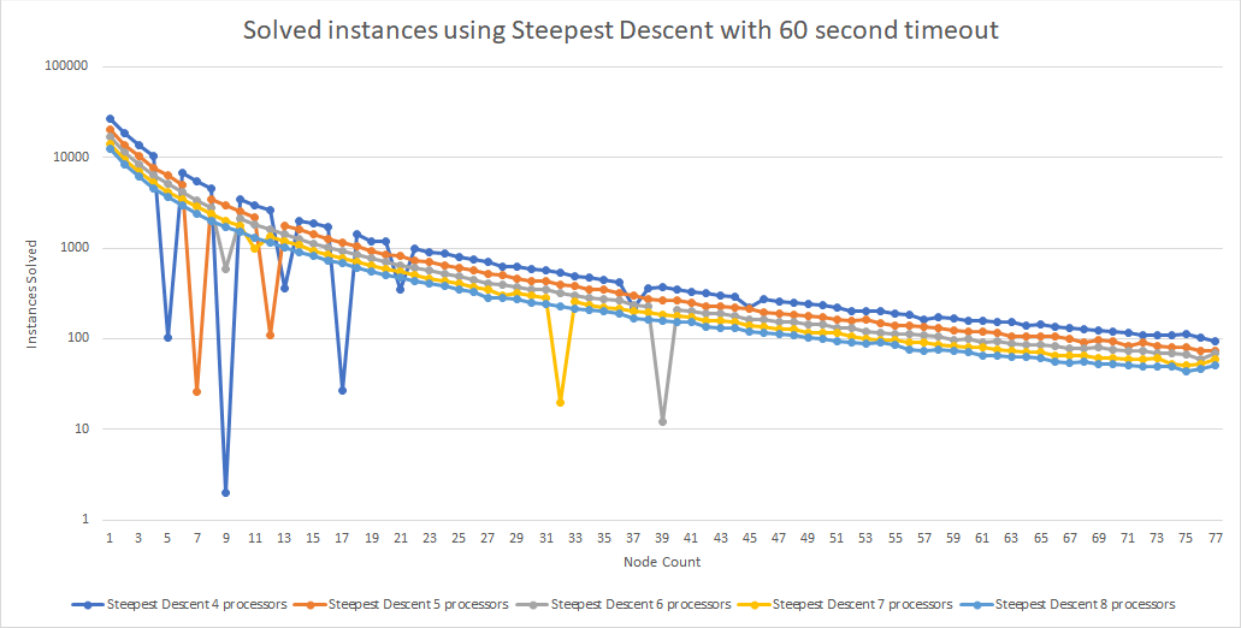


Fig. 2 Steepest Descent benchmark performance

The first difference we can see is that Steepest Descent (in this case, with random initialization, but the results do not vary too much for greedy initialization) can solve about two orders of magnitude more instances that GA.

We can clearly notice that GA cannot solve high node count instances (of more than around 40 nodes) within the 60 second timeout, while Steepest Descent does not have a limit within the simulated ranges.

As expected, both GA and Steepest Descent, can solve less instances as the number of nodes and/or processors is increased, since these define the size of the instance. However, for instances with a small number of tasks (node count smaller than around 20): in Steepest Descent, duplicating the node count

produces a similar decrease in the number of instances solved as duplicating the number of processors, while in GA, duplicating the node count produces a decrease in the number of instances solved that is double than when duplicating the number of processors.

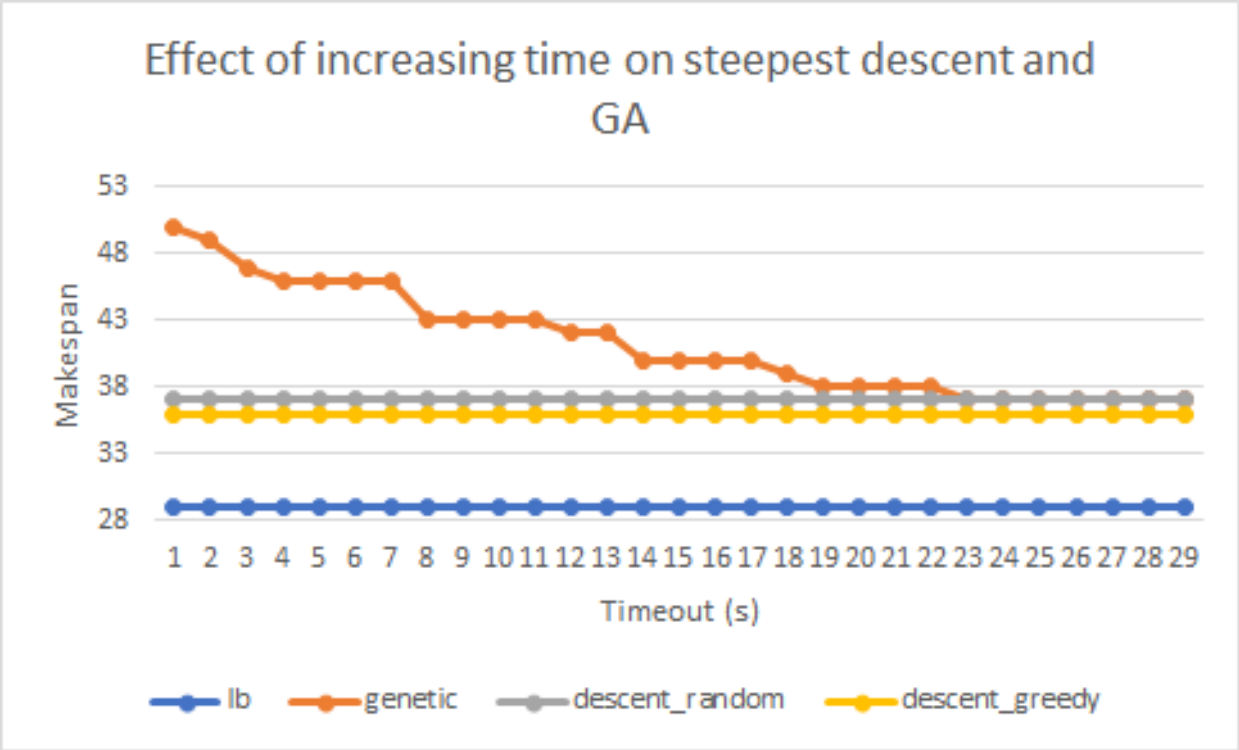


Fig. 3 Effect of timeout on GA and steepest descent

The configuration for figure 3 is as follows:

population_size = 25

cut_off = 15

mutation_rate = 0.30

fitness_tolerance = 0.8

max_steps_with_no_change = 20

The lower bound formulation and the steepest descent local search with random initialization provided solutions almost instantly for most of the instances in the benchmark. Steepest descent local search with greedy initialization took some more time for big instances with complex precedence relations, but still could compute them in one or two seconds. This is the reason why there is no effect on the makespan when increasing the timeout.

For the case of the genetic algorithm, as we can see, the makespan does decrease as the timeout is increased. However, we can notice a limit in the makespan, which does not improve beyond the steepest descent result. This is due to a limitation in the scheduling function that we used, which skews towards a greedy allocation and not the most optimal schedule.

**Solution quality**

As indicated by figures 4 through 9, due to the limitation on the scheduling algorithm, steepest descent and GA tend to converge to the greedy algorithm's solution regardless of instance size or processor count. Additionally, as mentioned in previous iterations of this report, the quality of the greedy solution increases with more processors. Given the convergence relationship between GA and steepest descent the quality of both improves greatly with more processors. Generally, with the current known limitation of the scheduling algorithm, the ILP model is the most useful technique in terms of solution quality in both cases where an optimal or a sub optimal can be found. However, when solving instances with no ILP solutions or exact solutions available greedy or steepest descent are the preferred techniques.

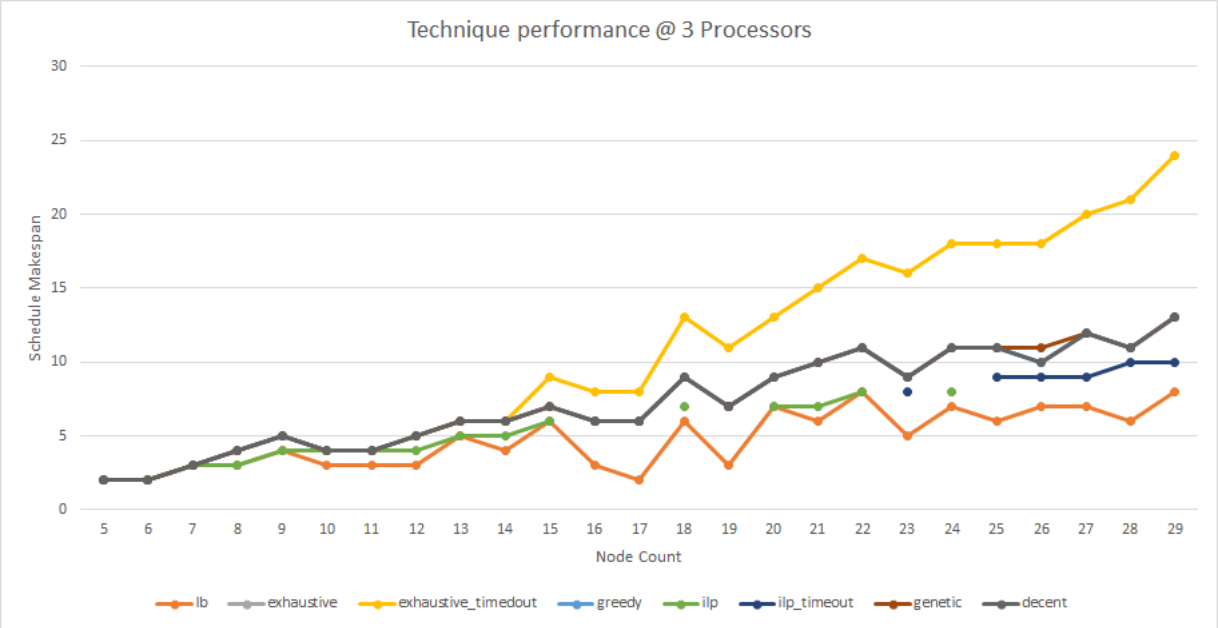


Fig 4. All technique's performance with 3 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 2.56

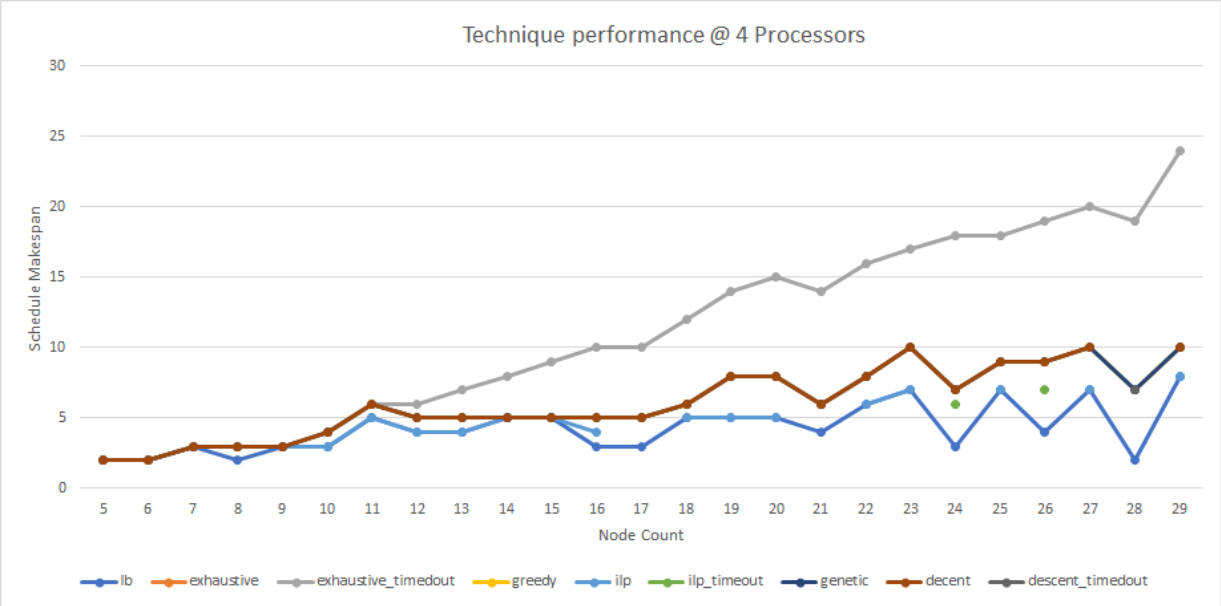


Fig 5. All technique's performance with 4 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 1.76

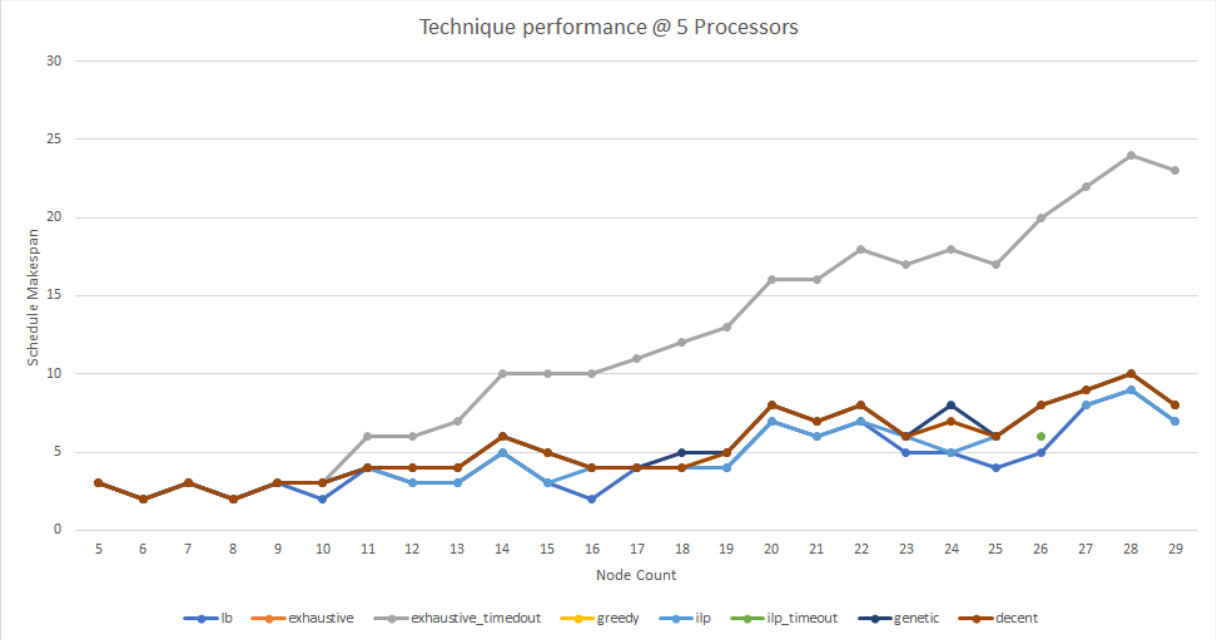


Fig 6. All technique's performance with 5 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 0.92

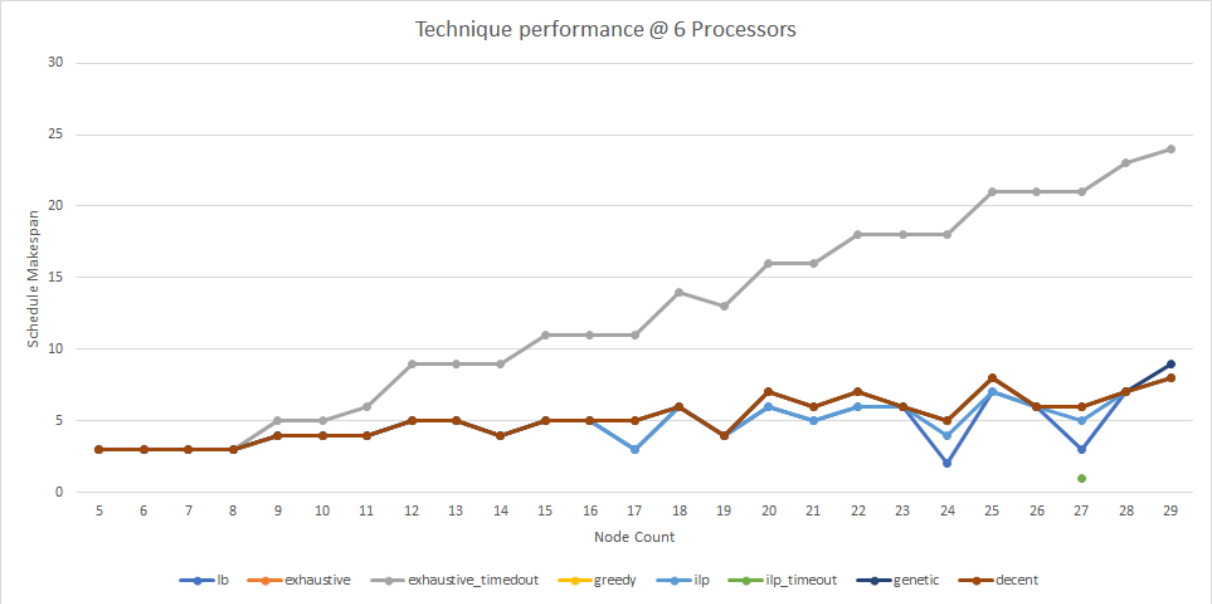


Fig 7. All technique's performance with 6 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 0.48

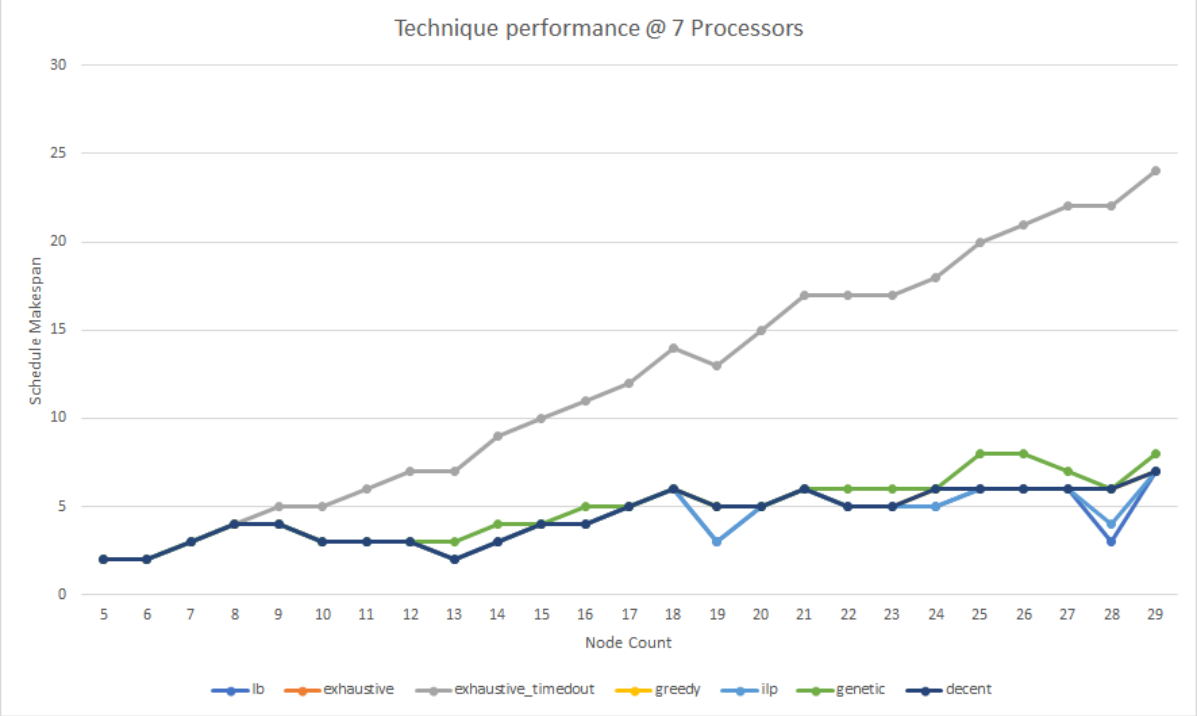


Fig 8. All technique's performance with 7 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 0.24

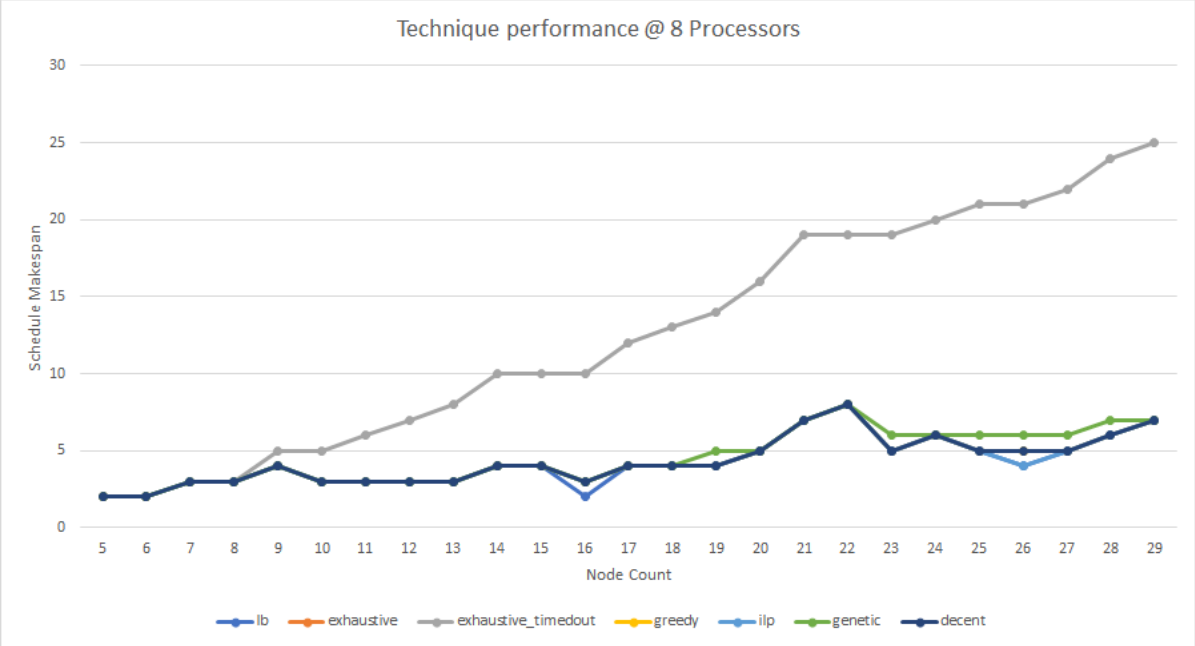


Fig 9. All technique's performance with 8 processors against previous techniques @ timeout = 10 seconds with average greedy-lb gap = 0.08

**Benchmark performance**

Figures 10 through 12 in combination with figures 1 and 2 provide a useful guide for which technique to choose depending on which type of instance you're trying to solve. Generally greedy is the fastest and exhaustive is the slowest. If an optimal solution is required and the sparsity of the graph is low ILP may be a favorable technique if the instance size permits it. If the number of processors is high enough all techniques converge and greedy becomes the most favorable. Assuming the scheduling bug isn't an issue and GA and steepest descent can converge to lower makespans below greedy a direct tradeoff between quality of results and time becomes available.

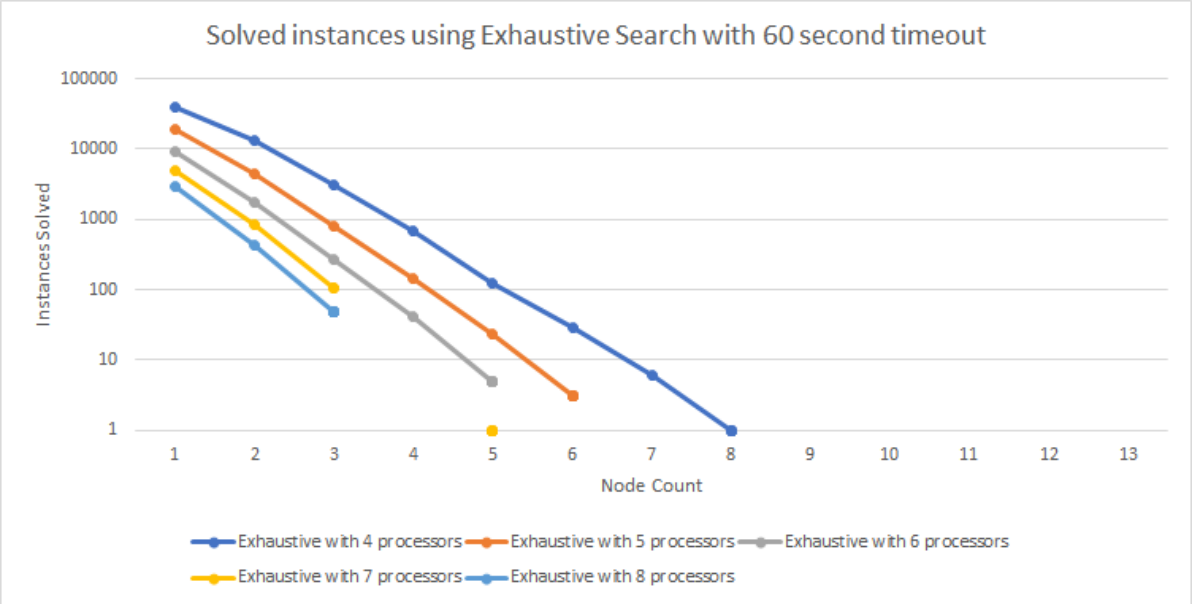


Fig. 10 Exhaustive search benchmark performance

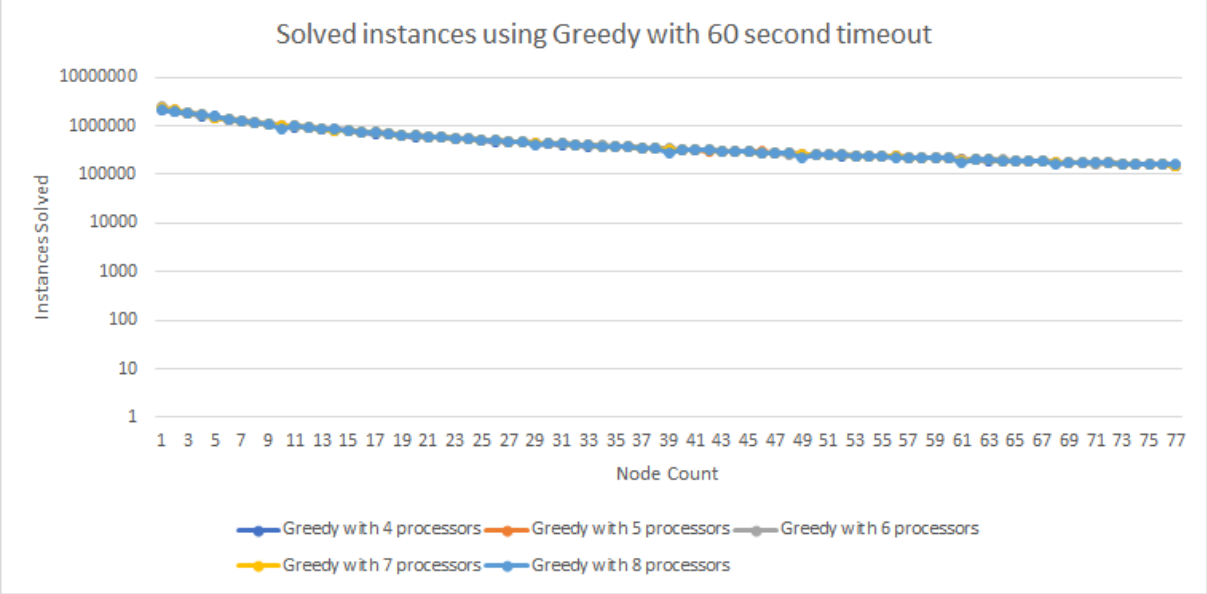


Fig. 11 Greedy benchmark performance

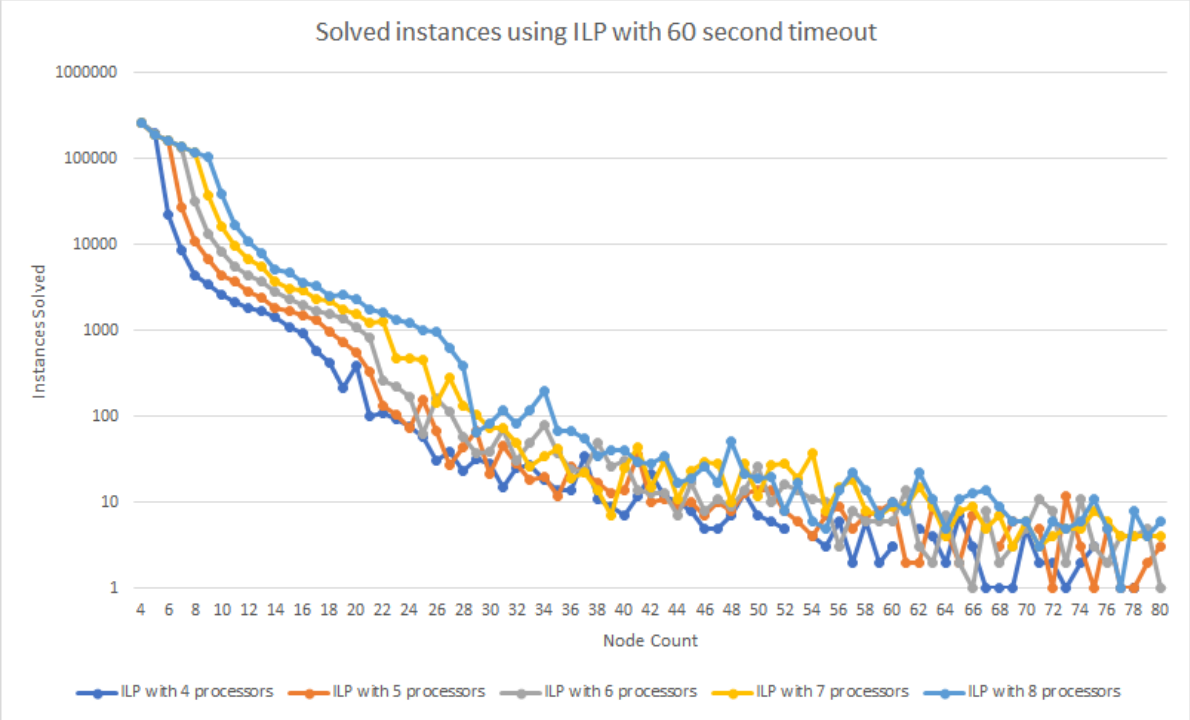


Fig. 12 ILP benchmark performance

References

[1]E. Aarts and J. K. Lenstra, Eds., “Local Search in Combinatorial Optimization.” Princeton University Press, Jun. 05, 2018. doi: 10.2307/j.ctv346t9c.