NI-KOP 1. Domácí úloha

GSAT vs probSAT

Filip Kašpar

December 13, 2024

1 Executive Summary

1.1 Introduction:

Originální zadání:

Experimentálně srovnejte algoritmy GSAT a probSAT. Určete, který algoritmus dospěje rychleji (v menším počtu iterací) k řešení obtížných instancí 3-SAT v rozsahu 20-75 proměnných. Zdůvodněte použité metody a metriky, popište interpretaci dat.

Uvažujte pevné parametry: 1. GSAT: p = 0.4 2. probSAT: cm = 0; cb = 2.3

Cílem práce je tedy určit, který z těchto algoritmů řeší 3SAT problémy s danými parametry efektivněji (za méně iterací).

1.2 Material:

Experimentální srovnání probíhalo následovně:

1. Získání splnitelných instancí 3-SAT problémů
2. Získání průměrného počtu iterací pro algoritmus GSAT a probSAT na různě těžkých 3-SAT problémech
3. Detailnější měření pro 20 nejobtížnějších instancí z každé sady
4. Průměrný počet iterací jedné instance na celou sadu
5. Vykreslení získaných dat pomocí distribuční funkce
6. Určení efektivnějšího algoritmu

Výše zmíněné kroky jsou následně podrobněji rozepsány níže ve stejném pořadí.

Experiment byl prováděn na vlastním stroji s použitím programovací jazyka Python.

1.3 Results:

Algoritmus probSAT je efektivnější než algoritmus GSAT.

Všechna naměřená data se nacházejí ve složce results.

1.4 Discussion:

Pro každou sadu 20, 36, 50 a 75 proměnných nabývala CDF hodnoty 1 rychleji (za menší počet iterací) v případě algoritmu probSAT než algoritmus GSAT. To tedy znamená, že algortimus probSAT

trvá kratší dobu na vyřešení 3-SAT problémů a tedy je rychlejší než algoritmus GSAT. Všechny grafy jsou k vidění v 5. bodě.

2 Získání splnitelných instancí 3-SAT problémů

Instance jsem získal ze stránek předmětu NI-KOP a to konkrétně zde: https://courses.fit.cvut.cz/NI-KOP/download/index.html

Zde jsem získal instance pro 3-SAT problémy o 20, 36, 50 a 75 proměnných. Každá instance ze sady je zároveň blízko fázového přechodu.

Sady s instancemi problému se jmenují následovně:

1. ruf20-91
2. ruf36-157
3. ruf50-218
4. ruf75-320

Kde první číslo označuje počet proměnných a druhé počet klauzulí.

Jelikož má každá sada 1000 instancí 3-SAT problémů, tak pro naše účely jsou tyto 4 sady dat dostačující.

3 Získání průměrného počtu iterací pro algoritmus GSAT a probSAT na různě těžkých 3-SAT instancích

Každá sada obsahuje 1000 instačních problémů. Pro každou sadu tedy spustíme několikrát každou instanci pro daný algoritmus a zjistíme kolik bylo potřeba průměrně iterací na vyřešení dané instance. Pro zjištění průměrného počtu iterací na danou sadu jednoduše sečteme celkový počet iterací v dané sadě a vydělíme počtem instancí.

Nejprve si definujme balíčky a konstanty se kterými bude program pracovat:

Vyhodnocení instance v sadě jsem opakoval několikrát, abych potlačil randomizaci. Proměnná FILE_REPEATS udává počet spuštění dané instance. Implementovaná metoda tedy nedává spolehlivé výsledky v jednom kroku, ale každou instanci opakuje několikrát.

[3]:
```
import subprocess
import numpy as np
import matplotlib.pyplot as plt
import re
import scipy.stats as st
```

[4]:
```
NUM_RUNS = 1
MAX_ITERATIONS = 100000000
FILES_IN_SET = 1000

# How many times is each instance in set being run
FILE_REPEATS = 1000
ITERATION_HIT_MULTIPLIER = 5
```

```
NUM_TOP_INSTANCES = 20
SAVE_FOLDER = "results/top"

GSAT_PROBABILITY = 0.4

PROBSAT_CM = 0
PROBSAT_CB = 2.3


# ( variables count, clauses count )
problem_files = [
    (20, 91),
    (36, 157),
    (50, 218),
    (75, 320),
]
```

Následně definujeme funkce, které spouští 3-SAT řešící algoritmy:

[15]:
```
def run_gsat(file_name_tup):
    process = subprocess.run(["./gsat2", "-p", f"{GSAT_PROBABILITY}", "-T",␣
↪f"{NUM_RUNS}", "-i", f"{MAX_ITERATIONS}", "-r", "time", f"problems/
↪ruf{file_name_tup[0]}-{file_name_tup[1]}/
↪ruf{file_name_tup[0]}-{file_name_tup[1]}-{instance_index+1}.cnf"],
                                                stderr=subprocess.PIPE,  #␣
↪Capture stderr
                                                stdout=subprocess.DEVNULL,  #␣
↪Ignore stdout
                                                text=True)

    return int(process.stderr.splitlines()[0].split()[0])
```

[16]:
```
def run_probsat(file_name_tup):
    process = subprocess.run(["./probSAT", "--fct", f"{PROBSAT_CM}", "--cb",␣
↪f"{PROBSAT_CB}", "--runs", f"{NUM_RUNS}", "--maxflips", f"{MAX_ITERATIONS}",␣
↪f"problems/ruf{file_name_tup[0]}-{file_name_tup[1]}/
↪ruf{file_name_tup[0]}-{file_name_tup[1]}-{instance_index+1}.cnf"],
                                               stdout=subprocess.PIPE,  # Capture␣
↪stdout
                                               stderr=subprocess.DEVNULL,  # Ignore␣
↪stderr
                                               text=True)

    return int(re.search(r'numFlips\s+:\s+(\d+)', process.stdout).group(1))
```

Algoritmus probSAT byl převzat z githubu od autora nápadu algoritmu probSAT. https://github.com/adrianopolus/probSAT

Algoritmus byl lehce modifikován, tak aby generoval seed náhodného generátoru na základě času v nanosekundách, oproti sekundám, jak bylo v původní implementaci.

[ ]:
```
algorithms = {
    "gsat" : run_gsat,
    "probsat" : run_probsat,
}
```

Hlavní funkce, která zkoumá rychlost jednotlivých instancí v daných sadách a následně informace ukládá, potom vypadá následovně:

[ ]:
```
for algorithm_name, algorithm_func in algorithms.items():
    for file_name_tup in problem_files:
        files_in_set = {}
        for instance_index in range(FILES_IN_SET):
            total_iteration_file = 0
            for _ in range(FILE_REPEATS):
                iterations = algorithm_func(file_name_tup)

                # In case the instance hasn't been resolved in time more␣
↪iterations are added
                if iterations == MAX_ITERATIONS:
                    total_iteration_file += MAX_ITERATIONS *␣
↪ITERATION_HIT_MULTIPLIER
                    print("Iteration hit!")
                else:
                    total_iteration_file += iterations

            # Average iterations per file
            total_iteration_file /= FILE_REPEATS


          ␣
↪files_in_set[f"ruf{file_name_tup[0]}-{file_name_tup[1]}-{instance_index+1}.
↪cnf"] = total_iteration_file
            print(f"File␣
↪ruf{file_name_tup[0]}-{file_name_tup[1]}-{instance_index+1}.cnf done! Using␣
↪{algorithm_name}")

        # Sort instances based on iterations from highest to lowest
        sorted_files_in_set = dict(sorted(files_in_set.items(), key=lambda item:␣
↪item[1], reverse=True))

        with open(f"results/all/{algorithm_name}/
↪ruf{file_name_tup[0]}-{file_name_tup[1]}.txt", "w") as file:
            for key, value in sorted_files_in_set.items():
                file.write(f"{key} : {value} iterations\n")
```

```
        print(f"Calculations for ruf{file_name_tup[0]}-{file_name_tup[1]} saved!
↪")
```

Výsledné průměry jednotlivých instancí jsou k vidění ve složkách results/all/gsat a results/all/probsat

4 Detailnější měření pro 20 nejobtížnějších instancí z každé sady

Pro každou sadu a algoritmus byl vytvořen soubor, který řadí instance z dané sady od nejtěžších po nejlehčí (z hlediska počtu iterací). Tedy nejnáročnější instance se nacházejí na prvních řádcích příslušných souborů/sad. Použijeme tedy z každé sady 20 nejtěžších instancí, abychom naměřili pro oba algoritmy jejich průměrnou dobu na vyřešení instance.

Nejtěžší instance vezmeme průnikem z výsledků nejtěžších instancí sad z algoritmu GSAT a probSAT.

Nejtěžší instance z každé sady, dle prvního měření.

[6]:
```
hardest_instances = {
    "ruf20-91": [],
    "ruf36-157": [],
    "ruf50-218": [],
    "ruf75-320": [],
}

NUM_TOP_INSTANCES = 20
```

Lehce modifikujeme algoritmus z druhého bodu, tak abychom počítali pouze 20 nejtěžších instancí, oproti celé sadě. Zejména zvedneme hodnotu proměnné FILE_REPEATS na 200 000. Tím zajistíme přesnější výsledky.

Výsledky výpočtů jsou k vidění ve složce results/top/

5 Průměrný počet iterací celé sady na jednu instanci

Tato kapitola je zde, jen kvůli tomu, že jsem se chtěl podívat kolik průměrně iterací bude daná sada potřebovat na vyřešení instancí z dané sady. Následující funkce načte výsledky sad a vypočte jejich průměr.

Jelikož každá instance v sadě může být různě těžká, tak tímto způsobem dostaneme průměrnou náročnost dané sady daným algoritmem.

[6]:
```
algorithms_data = {
    "gsat" : {},
    "probsat" : {},
}

for algorithm_name, algorithm_dict in algorithms_data.items():
    for file_name_tup in problem_files:
        with open(f"results/top/{algorithm_name}/
↪ruf{file_name_tup[0]}-{file_name_tup[1]}.txt", "r") as file:
            for line in file:
                value = float(re.search(r":\s*([\d.]+)\s*iterations", line).
↪group(1))
                algorithms_data[f"{algorithm_name}"].
↪setdefault(f"ruf{file_name_tup[0]}-{file_name_tup[1]}", []).append(value)
```

Pomocí následující funkce zjistíme průměrný počet iterací na danou sadu a algoritmus:

[8]:
```
for file_name_tup in problem_files:
    set_iterations = 0
    num_instances_in_set = 0
    for algorithm_name, algorithm_dict in algorithms_data.items():
        for iterations in␣
↪algorithms_data[f"{algorithm_name}"][f"ruf{file_name_tup[0]}-{file_name_tup[1]}"]:
↪
            set_iterations += iterations
            num_instances_in_set += 1

        print(f"{algorithm_name} ruf{file_name_tup[0]}-{file_name_tup[1]} set␣
↪has average of {set_iterations / num_instances_in_set} iterations.")
```

```
gsat ruf20-91 set has average of 184.03816375000002 iterations.
probsat ruf20-91 set has average of 139.80908825 iterations.
gsat ruf36-157 set has average of 557.801897 iterations.
probsat ruf36-157 set has average of 415.176395625 iterations.
gsat ruf50-218 set has average of 3837.4281645 iterations.
probsat ruf50-218 set has average of 2479.4080026250003 iterations.
gsat ruf75-320 set has average of 2991.5972319999996 iterations.
probsat ruf75-320 set has average of 2142.6466648749997 iterations.
```

Z výsledků to vypadá, že algoritmus probSAT je rychlejší (provede méně iterací) na všech instancích s 20, 36, 50 a 75 proměnnými.

6 Vykreslení získaných dat pomocí distribuční funkce

Nekorigované CDF křivky algoritmů můžeme porovnávat, pokud:

1. Provádíme testování nad stejnou množinou instancí,
2. Máme shodné nastavení pro testování pro oba algoritmy,
3. Stejné výstupy obou algoritmů (suma sloupců obou histogramů u obou algoritmů je stejná).

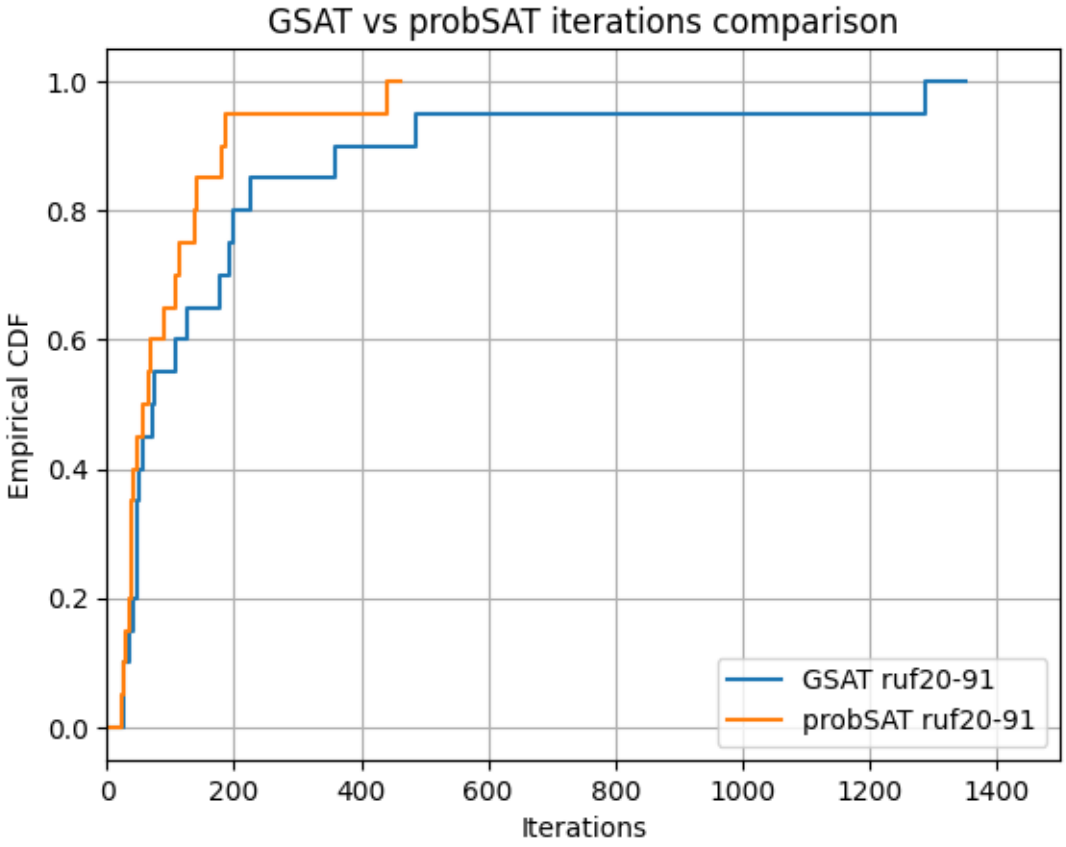
V našem případě to naše CDF splňují, tedy je můžeme porovnávat a dělat z nich závěry.

[9]:
```
def compare_cdf():
    fig, ax = plt.subplots()

    for algorithm_name, _ in algorithms_data.items():
        for gsat_name, gsat_data in algorithms_data[f"{algorithm_name}"].items():
            st.ecdf(gsat_data).cdf.plot(ax=ax, label=f"{algorithm_name}␣
↪{gsat_name}")

    ax.set_xlabel('Iterations')
    ax.set_ylabel('Empirical CDF')
    ax.set_title('GSAT vs probSAT iterations comparison')

    plt.xlim(0, XLIMIT)
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()
```

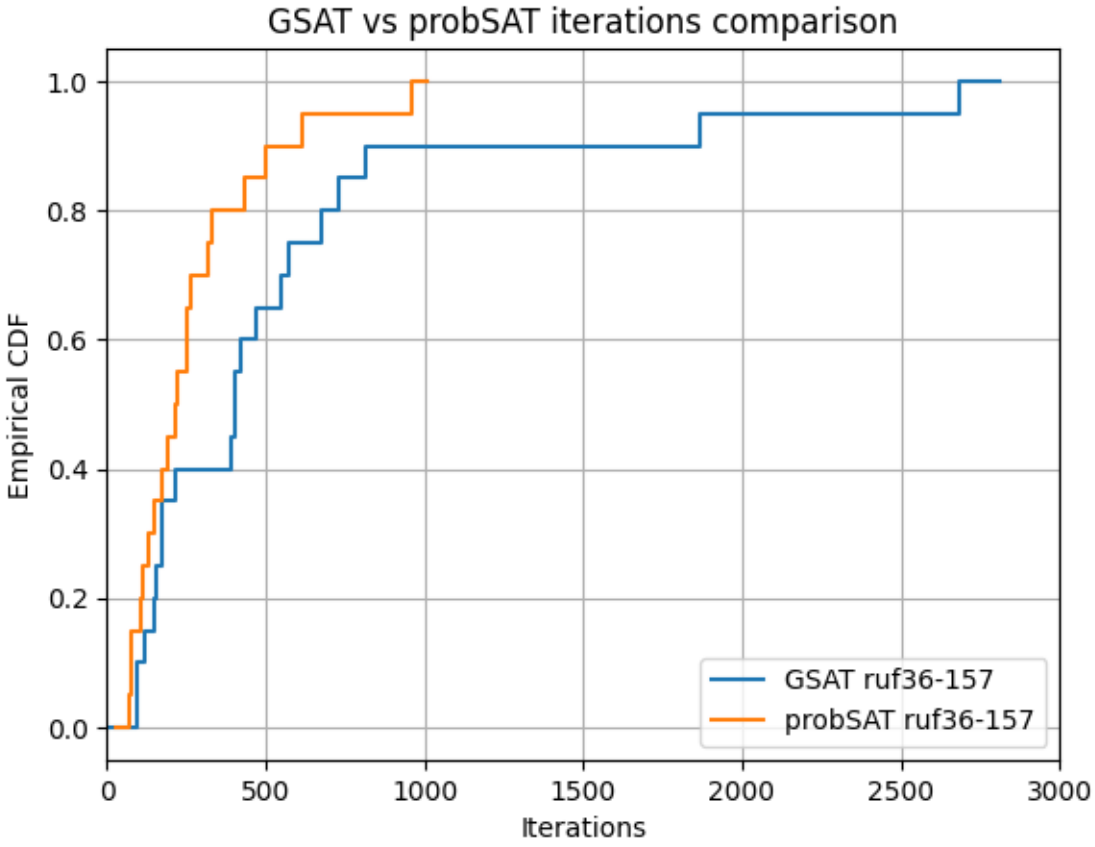
Pro srovnání pouze dvou cdf křivek využijeme funkci níže:

[8]:
```
def graph_cdf():
    fig, ax = plt.subplots()

    res1.cdf.plot(ax=ax, label=f"GSAT {CURVE_LABEL1}")
    res2.cdf.plot(ax=ax, label=f"probSAT {CURVE_LABEL2}")

    ax.set_xlabel('Iterations')
    ax.set_ylabel('Empirical CDF')
    ax.set_title('GSAT vs probSAT iterations comparison')

    plt.xlim(0, XLIMIT)
    plt.legend(loc="lower right"
    plt.grid(True)
    plt.show()
```

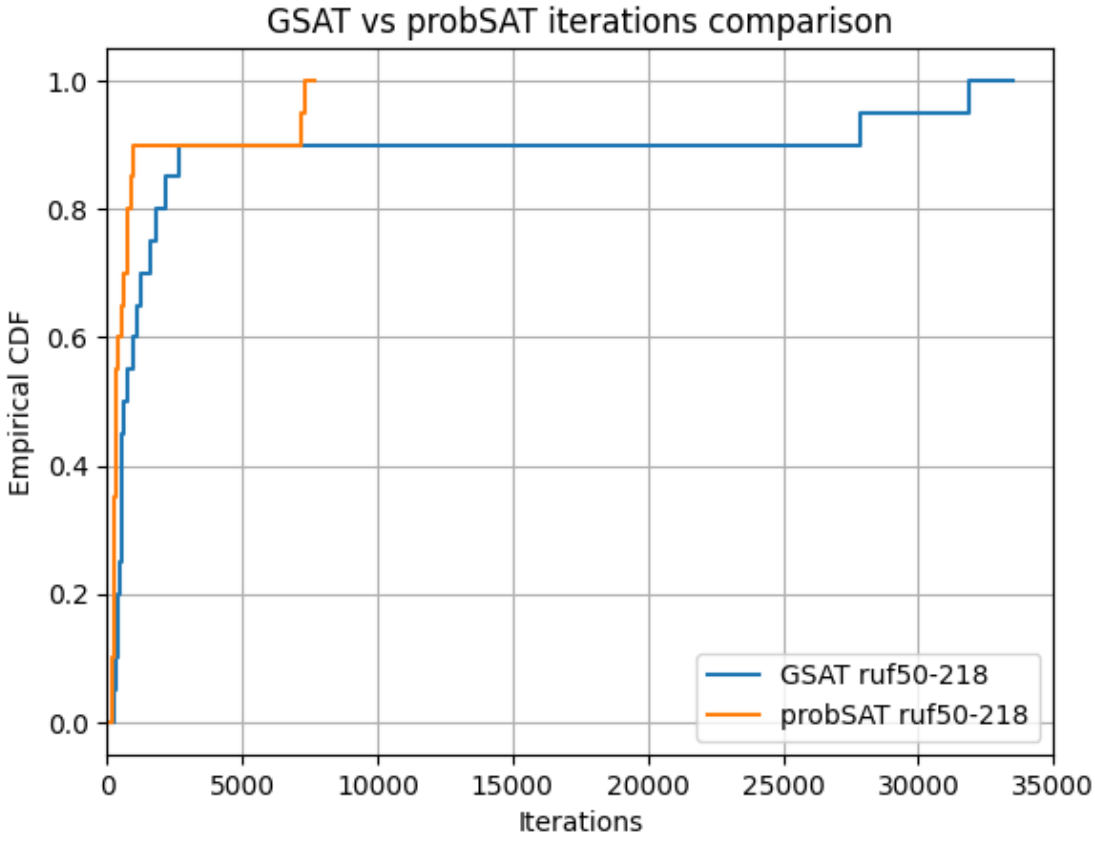
[25]:
```
CURVE_LABEL1 = "ruf20-91"
CURVE_LABEL2 = CURVE_LABEL1
res1 = st.ecdf(algorithms_data["gsat"][f"{CURVE_LABEL1}"])
res2 = st.ecdf(algorithms_data["probsat"][f"{CURVE_LABEL2}"])
XLIMIT = 1500
graph_cdf()
```



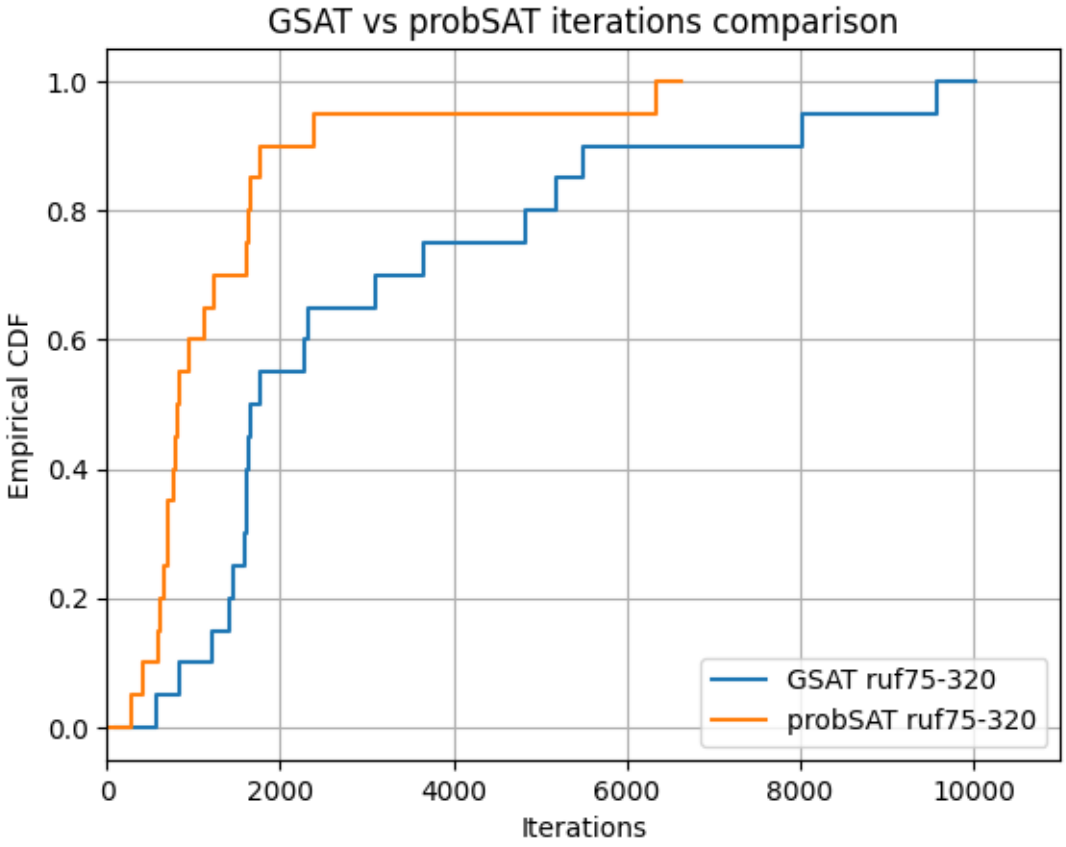
[20]:
```
CURVE_LABEL1 = "ruf36-157"
CURVE_LABEL2 = CURVE_LABEL1
res1 = st.ecdf(algorithms_data["gsat"][f"{CURVE_LABEL1}"])
res2 = st.ecdf(algorithms_data["probsat"][f"{CURVE_LABEL2}"])
XLIMIT = 3000
graph_cdf()
```



[15]:
```
CURVE_LABEL1 = "ruf50-218"
CURVE_LABEL2 = CURVE_LABEL1
res1 = st.ecdf(algorithms_data["gsat"][f"{CURVE_LABEL1}"])
res2 = st.ecdf(algorithms_data["probsat"][f"{CURVE_LABEL2}"])
XLIMIT = 35000
graph_cdf()
```



[18]:
```
CURVE_LABEL1 = "ruf75-320"
CURVE_LABEL2 = CURVE_LABEL1
res1 = st.ecdf(algorithms_data["gsat"][f"{CURVE_LABEL1}"])
res2 = st.ecdf(algorithms_data["probsat"][f"{CURVE_LABEL2}"])
XLIMIT = 11000
graph_cdf()
```



Získaná data a následné distribuční funkce jsou korektní a můžeme tedy z grafů usoudit, který z algoritmů je rychlejší.

7 Určení efektivnějšího algoritmu

Na základě získaných dat a následné analýzy je zřejmé, jak už z jednotlivých souborů výsledků sad, či CDF grafů, že algoritmus probSAT je rychlejší než algoritmus GSAT.