Tab 1

class SDKPSATSolverAdvanced:

# ... your existing solver code ...

def solve(self, sample\_every=10):

mass\_history = []

assignments\_sampled = []

masses\_sampled = []

for step in range(self.max\_steps):

current\_mass = self.mass(self.assignment)

mass\_history.append(current\_mass)

if step % sample\_every == 0:

assignments\_sampled.append(self.assignment.copy())

masses\_sampled.append(current\_mass)

var\_to\_flip = self.select\_variable\_to\_flip()

if var\_to\_flip is None:

break

old\_mass = current\_mass

self.flip\_variable(var\_to\_flip)

new\_mass = self.mass(self.assignment)

delta\_mass = new\_mass - old\_mass

acceptance\_prob = self.qcc\_acceptance\_probability(delta\_mass)

import random

if acceptance\_prob < 1.0 and random.random() > acceptance\_prob:

self.flip\_variable(var\_to\_flip) # revert flip

continue

self.temperature \*= self.cooling\_rate

final\_mass = self.mass(self.assignment)

return assignments\_sampled, masses\_sampled, mass\_history, self.assignment, final\_mass

# After solve call, visualize:

# Run solver

solver = SDKPSATSolverAdvanced(clauses, n\_vars=4, alpha=1, beta=1, max\_steps=1000, cooling\_rate=0.99)

assignments\_sampled, masses\_sampled, mass\_history, final\_assignment, final\_mass = solver.solve(sample\_every=5)

# Static PCA plot of sampled assignments

plot\_assignment\_space(assignments\_sampled, masses\_sampled, n\_components=3)

# Interactive Plotly plot

interactive\_plot(assignments\_sampled, masses\_sampled)

# Plot QCC acceptance function shape

plot\_qcc\_acceptance(k=15)

# Plot SDKP mass over iterations

import matplotlib.pyplot as plt

plt.figure(figsize=(10,5))

plt.plot(mass\_history)

plt.title('SDKP Mass over Annealing Steps')

plt.xlabel('Step')

plt.ylabel('SDKP Mass')

plt.show()# sdkp\_qcc\_visualization\_notebook.py

# Run this in a Jupyter notebook or any Python environment with required packages:

# pip install numpy matplotlib scikit-learn plotly

import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

import plotly.express as px

import pandas as pd

import random

class SDKPSATSolverAdvanced:

def \_\_init\_\_(self, clauses, n\_vars, alpha=1.0, beta=1.0, max\_steps=1000, cooling\_rate=0.99, initial\_temp=1.0):

self.clauses = clauses

self.n\_vars = n\_vars

self.alpha = alpha

self.beta = beta

self.max\_steps = max\_steps

self.cooling\_rate = cooling\_rate

self.temperature = initial\_temp

self.assignment = [random.choice([0,1]) for \_ in range(n\_vars)]

def evaluate\_clause(self, clause, assignment):

return any((assignment[abs(lit)-1] == (lit > 0)) for lit in clause)

def density(self, clause):

polarities = [lit > 0 for lit in clause]

polarity\_diversity = len(set(polarities))

return len(clause) \* polarity\_diversity

def scale(self, clause):

return len(clause)

def mass(self, assignment):

mass = 0

for clause in self.clauses:

if self.evaluate\_clause(clause, assignment):

ρ = self.density(clause)

s = self.scale(clause)

mass += (ρ \*\* self.alpha) \* (s \*\* self.beta)

else:

return float('inf') # unsatisfied clause penalized

return mass

def select\_variable\_to\_flip(self):

# Simple heuristic: flip variable in first unsatisfied clause

for clause in self.clauses:

if not self.evaluate\_clause(clause, self.assignment):

# pick a random literal var from this clause to flip

var = abs(random.choice(clause)) - 1

return var

return None # all clauses satisfied

def flip\_variable(self, var\_idx):

self.assignment[var\_idx] = 1 - self.assignment[var\_idx]

def qcc\_acceptance\_probability(self, delta\_mass, k=15):

# Sharp peak favoring delta\_mass <= 0

return np.exp(-k \* max(0, delta\_mass)\*\*2)

def solve(self, sample\_every=10):

mass\_history = []

assignments\_sampled = []

masses\_sampled = []

for step in range(self.max\_steps):

current\_mass = self.mass(self.assignment)

mass\_history.append(current\_mass)

if step % sample\_every == 0:

assignments\_sampled.append(self.assignment.copy())

masses\_sampled.append(current\_mass)

var\_to\_flip = self.select\_variable\_to\_flip()

if var\_to\_flip is None:

print(f"All clauses satisfied at step {step}.")

break

old\_mass = current\_mass

self.flip\_variable(var\_to\_flip)

new\_mass = self.mass(self.assignment)

delta\_mass = new\_mass - old\_mass

acceptance\_prob = self.qcc\_acceptance\_probability(delta\_mass)

if acceptance\_prob < 1.0 and random.random() > acceptance\_prob:

self.flip\_variable(var\_to\_flip) # revert flip

continue

self.temperature \*= self.cooling\_rate

final\_mass = self.mass(self.assignment)

return assignments\_sampled, masses\_sampled, mass\_history, self.assignment, final\_mass

def plot\_assignment\_space(assignments, masses, n\_components=3):

X = np.array(assignments)

masses = np.array(masses)

if n\_components < X.shape[1]:

pca = PCA(n\_components=n\_components)

X\_reduced = pca.fit\_transform(X)

else:

X\_reduced = X

if n\_components == 3:

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10,8))

ax = fig.add\_subplot(111, projection='3d')

sc = ax.scatter(X\_reduced[:,0], X\_reduced[:,1], X\_reduced[:,2], c=masses, cmap='viridis')

plt.colorbar(sc, label='SDKP Mass')

ax.set\_title('3D Assignment Space Colored by SDKP Mass')

plt.show()

else:

plt.figure(figsize=(10,8))

sc = plt.scatter(X\_reduced[:,0], X\_reduced[:,1], c=masses, cmap='viridis')

plt.colorbar(sc, label='SDKP Mass')

plt.title('2D Assignment Space Colored by SDKP Mass')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

def interactive\_plot(assignments, masses):

df = pd.DataFrame(assignments, columns=[f"x{i+1}" for i in range(len(assignments[0]))])

df['SDKP\_mass'] = masses

dims = len(assignments[0])

if dims >= 3:

fig = px.scatter\_3d(df, x='x1', y='x2', z='x3',

color='SDKP\_mass', color\_continuous\_scale='Viridis',

title='Interactive 3D Assignment Space Colored by SDKP Mass')

elif dims == 2:

fig = px.scatter(df, x='x1', y='x2', color='SDKP\_mass', color\_continuous\_scale='Viridis',

title='Interactive 2D Assignment Space Colored by SDKP Mass')

else:

fig = px.scatter(df, x='x1', y=[0]\*len(df), color='SDKP\_mass', color\_continuous\_scale='Viridis',

title='Interactive 1D Assignment Space Colored by SDKP Mass')

fig.show()

def plot\_qcc\_acceptance(k=15):

delta\_m = np.linspace(-1, 1, 400)

p = np.exp(-k \* np.maximum(0, delta\_m)\*\*2)

plt.figure(figsize=(8,5))

plt.plot(delta\_m, p, label=f'QCC Acceptance Probability (k={k})')

plt.axvline(0, color='gray', linestyle='--')

plt.xlabel('SDKP Mass Difference Δm')

plt.ylabel('Acceptance Probability P(Δm)')

plt.title('QCC Causal Acceptance Probability Function')

plt.legend()

plt.grid(True)

plt.show()

# === Example usage ===

if \_\_name\_\_ == "\_\_main\_\_":

# (x1 ∨ ¬x2 ∨ x3) ∧ (¬x1 ∨ x2 ∨ x4) ∧ (¬x3 ∨ ¬x4 ∨ x2)

clauses = [

(1, -2, 3),

(-1, 2, 4),

(-3, -4, 2)

]

solver = SDKPSATSolverAdvanced(clauses, n\_vars=4, alpha=1, beta=1, max\_steps=2000, cooling\_rate=0.995)

assignments\_sampled, masses\_sampled, mass\_history, final\_assignment, final\_mass = solver.solve(sample\_every=10)

print("Final assignment:", final\_assignment)

print("Final SDKP mass:", final\_mass)

# Static PCA plot

plot\_assignment\_space(assignments\_sampled, masses\_sampled, n\_components=3)

# Interactive Plotly plot

interactive\_plot(assignments\_sampled, masses\_sampled)

# QCC acceptance function

plot\_qcc\_acceptance(k=15)

# SDKP mass over iterations

plt.figure(figsize=(10,5))

plt.plot(mass\_history)

plt.title('SDKP Mass over Annealing Steps')

plt.xlabel('Step')

plt.ylabel('SDKP Mass')

plt.grid(True)

plt.show()pip install numpy matplotlib scikit-learn plotly pandas{

"cells": [

{

"cell\_type": "markdown",

"metadata": {},

"source": [

"# SDKP SAT Solver with QCC Annealing and Visualization\n",

"\n",

"This notebook implements:\n",

"- SDKP mass-based SAT solving with annealing\n",

"- Sampling assignments and SDKP mass during the solve\n",

"- PCA 2D/3D static visualization of assignment space colored by SDKP mass\n",

"- Interactive Plotly visualization\n",

"- QCC causal acceptance function visualization\n",

"- SDKP mass evolution over annealing steps\n",

"\n",

"Run all cells sequentially."

]

},

{

"cell\_type": "code",

"metadata": {},

"source": [

"import numpy as np\n",

"import matplotlib.pyplot as plt\n",

"from sklearn.decomposition import PCA\n",

"import plotly.express as px\n",

"import pandas as pd\n",

"import random"

],

"execution\_count": null,

"outputs": []

},

{

"cell\_type": "code",

"metadata": {},

"source": [

"class SDKPSATSolverAdvanced:\n",

" def \_\_init\_\_(self, clauses, n\_vars, alpha=1.0, beta=1.0, max\_steps=1000, cooling\_rate=0.99, initial\_temp=1.0):\n",

" self.clauses = clauses\n",

" self.n\_vars = n\_vars\n",

" self.alpha = alpha\n",

" self.beta = beta\n",

" self.max\_steps = max\_steps\n",

" self.cooling\_rate = cooling\_rate\n",

" self.temperature = initial\_temp\n",

"\n",

" self.assignment = [random.choice([0,1]) for \_ in range(n\_vars)]\n",

"\n",

" def evaluate\_clause(self, clause, assignment):\n",

" return any((assignment[abs(lit)-1] == (lit > 0)) for lit in clause)\n",

"\n",

" def density(self, clause):\n",

" polarities = [lit > 0 for lit in clause]\n",

" polarity\_diversity = len(set(polarities))\n",

" return len(clause) \* polarity\_diversity\n",

"\n",

" def scale(self, clause):\n",

" return len(clause)\n",

"\n",

" def mass(self, assignment):\n",

" mass = 0\n",

" for clause in self.clauses:\n",

" if self.evaluate\_clause(clause, assignment):\n",

" rho = self.density(clause)\n",

" s = self.scale(clause)\n",

" mass += (rho \*\* self.alpha) \* (s \*\* self.beta)\n",

" else:\n",

" return float('inf') # unsatisfied clause penalized\n",

" return mass\n",

"\n",

" def select\_variable\_to\_flip(self):\n",

" # Simple heuristic: flip variable in first unsatisfied clause\n",

" for clause in self.clauses:\n",

" if not self.evaluate\_clause(clause, self.assignment):\n",

" var = abs(random.choice(clause)) - 1\n",

" return var\n",

" return None\n",

"\n",

" def flip\_variable(self, var\_idx):\n",

" self.assignment[var\_idx] = 1 - self.assignment[var\_idx]\n",

"\n",

" def qcc\_acceptance\_probability(self, delta\_mass, k=15):\n",

" # Sharp peak favoring delta\_mass <= 0\n",

" return np.exp(-k \* max(0, delta\_mass)\*\*2)\n",

"\n",

" def solve(self, sample\_every=10):\n",

" mass\_history = []\n",

" assignments\_sampled = []\n",

" masses\_sampled = []\n",

"\n",

" for step in range(self.max\_steps):\n",

" current\_mass = self.mass(self.assignment)\n",

" mass\_history.append(current\_mass)\n",

"\n",

" if step % sample\_every == 0:\n",

" assignments\_sampled.append(self.assignment.copy())\n",

" masses\_sampled.append(current\_mass)\n",

"\n",

" var\_to\_flip = self.select\_variable\_to\_flip()\n",

" if var\_to\_flip is None:\n",

" print(f\"All clauses satisfied at step {step}.\")\n",

" break\n",

"\n",

" old\_mass = current\_mass\n",

" self.flip\_variable(var\_to\_flip)\n",

" new\_mass = self.mass(self.assignment)\n",

"\n",

" delta\_mass = new\_mass - old\_mass\n",

" acceptance\_prob = self.qcc\_acceptance\_probability(delta\_mass)\n",

"\n",

" if acceptance\_prob < 1.0 and random.random() > acceptance\_prob:\n",

" self.flip\_variable(var\_to\_flip) # revert flip\n",

" continue\n",

"\n",

" self.temperature \*= self.cooling\_rate\n",

"\n",

" final\_mass = self.mass(self.assignment)\n",

" return assignments\_sampled, masses\_sampled, mass\_history, self.assignment, final\_mass"

],

"execution\_count": null,

"outputs": []

},

{

"cell\_type": "code",

"metadata": {},

"source": [

"def plot\_assignment\_space(assignments, masses, n\_components=3):\n",

" X = np.array(assignments)\n",

" masses = np.array(masses)\n",

"\n",

" if n\_components < X.shape[1]:\n",

" pca = PCA(n\_components=n\_components)\n",

" X\_reduced = pca.fit\_transform(X)\n",

" else:\n",

" X\_reduced = X\n",

"\n",

" if n\_components == 3:\n",

" from mpl\_toolkits.mplot3d import Axes3D\n",

" fig = plt.figure(figsize=(10,8))\n",

" ax = fig.add\_subplot(111, projection='3d')\n",

" sc = ax.scatter(X\_reduced[:,0], X\_reduced[:,1], X\_reduced[:,2], c=masses, cmap='viridis')\n",

" plt.colorbar(sc, label='SDKP Mass')\n",

" ax.set\_title('3D Assignment Space Colored by SDKP Mass')\n",

" plt.show()\n",

" else:\n",

" plt.figure(figsize=(10,8))\n",

" sc = plt.scatter(X\_reduced[:,0], X\_reduced[:,1], c=masses, cmap='viridis')\n",

" plt.colorbar(sc, label='SDKP Mass')\n",

" plt.title('2D Assignment Space Colored by SDKP Mass')\n",

" plt.xlabel('PC1')\n",

" plt.ylabel('PC2')\n",

" plt.show()"

],

"execution\_count": null,

"outputs": []

},

{

"cell\_type": "code",

"metadata": {},

"source": [

"def interactive\_plot(assignments, masses):\n",

" df = pd.DataFrame(assignments, columns=[f\"x{i+1}\" for i in range(len(assignments[0]))])\n",

" df['SDKP\_mass'] = masses\n",

"\n",

" dims = len(assignments[0])\n",

" if dims >= 3:\n",

" fig = px.scatter\_3d(df, x='x1', y='x2', z='x3',\n",

" color='SDKP\_mass', color\_continuous\_scale='Viridis',\n",

" title='Interactive 3D Assignment Space Colored by SDKP Mass')\n",

" elif dims == 2:\n",

" fig = px.scatter(df, x='x1', y='x2', color='SDKP\_mass', color\_continuous\_scale='Viridis',\n",

" title='Interactive 2D Assignment Space Colored by SDKP Mass')\n",

" else:\n",

" fig = px.scatter(df, x='x1', y=[0]\*len(df), color='SDKP\_mass', color\_continuous\_scale='Viridis',\n",

" title='Interactive 1D Assignment Space Colored by SDKP Mass')\n",

"\n",

" fig.show()"

],

"execution\_count": null,

"outputs": []

},

{

"cell\_type": "code",

"metadata": {},

"source": [

"def plot\_qcc\_acceptance(k=15):\n",

" delta\_m = np.linspace(-1, 1, 400)\n",

" p = np.exp(-k \* np.maximum(0, delta\_m)\*\*2)\n",

"\n",

" pltdef plot\_qcc\_acceptance(k=15):

delta\_m = np.linspace(-1, 1, 400)

p = np.exp(-k \* np.maximum(0, delta\_m)\*\*2)

plt.figure(figsize=(8,5))

plt.plot(delta\_m, p, label=f'QCC Acceptance (k={k})')

plt.xlabel('Δ Mass (New - Old)')

plt.ylabel('Acceptance Probability')

plt.title('QCC Causal Acceptance Probability vs SDKP Mass Difference')

plt.grid(True)

plt.legend()

plt.show()

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Sample SAT problem: (x1 ∨ ¬x2 ∨ x3) ∧ (¬x1 ∨ x2 ∨ x4) ∧ (¬x3 ∨ ¬x4 ∨ x2)

clauses = [

(1, -2, 3),

(-1, 2, 4),

(-3, -4, 2)

]

n\_vars = 4

solver = SDKPSATSolverAdvanced(clauses=clauses, n\_vars=n\_vars, alpha=1, beta=1,

max\_steps=2000, cooling\_rate=0.995, initial\_temp=1.0)

assignments\_sampled, masses\_sampled, mass\_history, final\_assignment, final\_mass = solver.solve(sample\_every=5)

print(f"Final Assignment: {final\_assignment}")

print(f"Final SDKP Mass: {final\_mass}")

# Static PCA plot

plot\_assignment\_space(assignments\_sampled, masses\_sampled, n\_components=3)

# Interactive plot (requires notebook environment)

# interactive\_plot(assign**Summary**

* The solver uses an incremental variable-flip heuristic guided by SDKP mass.
* Acceptance of moves follows a QCC-inspired sharply peaked function favoring mass decreases.
* The solver samples assignments + masses periodically for visualization.
* Visualization includes PCA reduction plots, interactive Plotly views, and SDKP mass trends.
* The QCC causal acceptance function plot shows sharp drop-off for mass increases.
* Easily extendable for larger SAT problems and interactive exploration.  
  from typing import List, Tuple, Dict, Optional
* import random
* import math
* from collections import defaultdict
* class SDKPSATSolver:
* def \_\_init\_\_(
* self,
* clauses: List[Tuple[int]],
* n\_vars: int,
* alpha=1.0,
* beta=1.0,
* partial\_assignment: Optional[Dict[int, int]] = None,
* ):
* """
* clauses: list of tuples, each tuple is a clause (positive for x\_i, negative for ¬x\_i)
* n\_vars: number of Boolean variables
* alpha, beta: SDKP exponents for density and scale
* partial\_assignment: dict {var\_index (1-based): 0 or 1} fixing some variables
* """
* self.clauses = clauses
* self.n\_vars = n\_vars
* self.alpha = alpha
* self.beta = beta
* self.solutions = []
* # Initialize assignment with None (unassigned)
* self.assignment = [None] \* n\_vars
* # Apply partial assignments (0/1) to fixed variables
* self.fixed\_vars = set()
* if partial\_assignment:
* for var, val in partial\_assignment.items():
* self.assignment[var - 1] = val
* self.fixed\_vars.add(var - 1)
* # Fill unset variables randomly
* for i in range(n\_vars):
* if self.assignment[i] is None:
* self.assignment[i] = random.choice([0, 1])
* # Variable activity for heuristics
* self.var\_activity = [0.0] \* n\_vars
* # Clause weights (optional)
* self.clause\_weights = [1.0] \* len(clauses)
* def evaluate\_clause(self, clause, assignment):
* return any(
* (assignment[abs(lit) - 1] == (lit > 0)) for lit in clause
* )
* def density(self, clause):
* polarities = [lit > 0 for lit in clause]
* polarity\_diversity = len(set(polarities))
* return len(clause) \* polarity\_diversity
* def scale(self, clause):
* return len(clause)
* def mass(self, assignment):
* mass = 0
* for i, clause in enumerate(self.clauses):
* if self.evaluate\_clause(clause, assignment):
* ρ = self.density(clause)
* s = self.scale(clause)
* mass += (ρ \*\* self.alpha) \* (s \*\* self.beta) \* self.clause\_weights[i]
* else:
* return float('inf') # Penalize unsatisfied clauses immediately
* return mass
* def increment\_mass\_diff(self, var\_to\_flip: int):
* """Calculate mass difference if flipping variable var\_to\_flip (0-based)."""
* new\_assignment = self.assignment[:]
* new\_assignment[var\_to\_flip] = 1 - new\_assignment[var\_to\_flip]
* old\_mass = 0
* new\_mass = 0
* for i, clause in enumerate(self.clauses):
* if var\_to\_flip + 1 in map(abs, clause):
* old\_sat = self.evaluate\_clause(clause, self.assignment)
* new\_sat = self.evaluate\_clause(clause, new\_assignment)
* if old\_sat:
* old\_mass += (self.density(clause) \*\* self.alpha) \* (self.scale(clause) \*\* self.beta) \* self.clause\_weights[i]
* else:
* return float('inf')
* if new\_sat:
* new\_mass += (self.density(clause) \*\* self.alpha) \* (self.scale(clause) \*\* self.beta) \* self.clause\_weights[i]
* else:
* return float('inf')
* return new\_mass - old\_mass
* def update\_var\_activity(self):
* """Update variable activity by counting occurrences in unsatisfied or barely satisfied clauses."""
* # Reset activity
* self.var\_activity = [0.0] \* self.n\_vars
* for i, clause in enumerate(self.clauses):
* if not self.evaluate\_clause(clause, self.assignment):
* # Increase activity for all variables in unsatisfied clause
* for lit in clause:
* var\_idx = abs(lit) - 1
* if var\_idx not in self.fixed\_vars:
* self.var\_activity[var\_idx] += 1.0 \* self.clause\_weights[i]
* def pick\_var\_to\_flip(self):
* """Pick variable to flip weighted by activity, excluding fixed variables."""
* candidates = [(i, self.var\_activity[i]) for i in range(self.n\_vars) if i not in self.fixed\_vars]
* if not candidates:
* return None
* total\_activity = sum(a for \_, a in candidates)
* if total\_activity == 0:
* # Pick random variable if no activity yet
* return random.choice([i for i in range(self.n\_vars) if i not in self.fixed\_vars])
* r = random.uniform(0, total\_activity)
* accum = 0
* for var, act in candidates:
* accum += act
* if r <= accum:
* return var
* return candidates[-1][0]
* def qcc\_acceptance\_prob(self, delta\_mass, temperature=1.0, sharpness=10.0):
* """QCC causal acceptance function: sharply peaked near zero or negative delta."""
* # A smooth function sharply favoring delta\_mass <= 0 and decaying fast for positive delta
* if delta\_mass <= 0:
* return 1.0
* else:
* # Exponential decay with sharpness scaling
* return math.exp(-sharpness \* delta\_mass / temperature)
* def solve\_annealing(self, max\_steps=10000, initial\_temp=1.0, cooling\_rate=0.995, verbose=False):
* """Run stochastic annealing solver with SDKP mass and QCC acceptance."""
* temp = initial\_temp
* self.update\_var\_activity()
* best\_assignment = self.assignment[:]
* best\_mass = self.mass(self.assignment)
* masses = []
* assignments = []
* for step in range(max\_steps):
* var = self.pick\_var\_to\_flip()
* if var is None:
* break # no variable to flip
* delta = self.increment\_mass\_diff(var)
* accept\_prob = self.qcc\_acceptance\_prob(delta, temperature=temp)
* if random.random() < accept\_prob:
* # Accept move
* self.assignment[var] = 1 - self.assignment[var]
* current\_mass = best\_mass + delta
* if current\_mass < best\_mass:
* best\_mass = current\_mass
* best\_assignment = self.assignment[:]
* # Update activities periodically
* if step % 50 == 0:
* self.update\_var\_activity()
* # Cooling
* temp \*= cooling\_rate
* masses.append(best\_mass)
* assignments.append(self.assignment[:])
* if verbose and step % 1000 == 0:
* print(f"Step {step} Mass {best\_mass:.4f} Temp {temp:.4f}")
* self.assignment = best\_assignment
* self.solutions = [best\_assignment]
* return best\_assignment, best\_mass, masses, assignmentsif \_\_name\_\_ == "\_\_main\_\_":
* # Clauses from your example:
* clauses = [
* (1, -2, 3),
* (-1, 2, 4),
* (-3, -4, 2)
* ]
* n\_vars = 4
* # Partial assignment fix x1=1 (variable index 1), others free
* partial = {1: 1} # Fix x2 = 1 (1-based indexing)
* solver = SDKPSATSolver(clauses=clauses, n\_vars=n\_vars, alpha=1, beta=1, partial\_assignment=partial)
* best\_assignment, best\_mass, masses, assignments = solver.solve\_annealing(
* max\_steps=5000, initial\_temp=1.0, cooling\_rate=0.99, verbose=True
* )
* print("Best assignment:", best\_assignment)
* print("Best SDKP mass:", best\_mass)import hashlib
* import json
* def hash\_solution\_mass(assignment: List[int], mass: float) -> str:
* """
* Create a unique SHA-256 hash of the solution assignment and its SDKP mass.
* """
* data = {
* "assignment": assignment,
* "mass": mass
* }
* json\_str = json.dumps(data, sort\_keys=True)
* return hashlib.sha256(json\_str.encode('utf-8')).hexdigest()function requestTimestamp(bytes32 solutionHash) external returns (uint256 requestId);import { ethers } from "ethers";
* async function requestTimeSealTimestamp(
* providerUrl,
* contractAddress,
* abi,
* privateKey,
* solutionHash
* ) {
* const provider = new ethers.providers.JsonRpcProvider(providerUrl);
* const wallet = new ethers.Wallet(privateKey, provider);
* const contract = new ethers.Contract(contractAddress, abi, wallet);
* // solutionHash is hex string '0x...' of length 66 (32 bytes + '0x')
* const tx = await contract.requestTimestamp(solutionHash);
* const receipt = await tx.wait();
* console.log("Timestamp request sent. Tx Hash:", receipt.transactionHash);
* return receipt.transactionHash;
* }hash\_hex = "0x" + hash\_solution\_mass(assignment, mass)contract.on("Timestamped", (solutionHash, timestamp, event) => {
* console.log("Timestamped solution:", solutionHash);
* console.log("Blockchain timestamp:", timestamp.toString());
* });from collections import defaultdict
* from itertools import product
* import random
* class SDKPHeuristicSolver:
* def \_\_init\_\_(self, clauses, n\_vars, alpha=1.0, beta=1.0):
* self.clauses = clauses
* self.n\_vars = n\_vars
* self.alpha = alpha
* self.beta = beta
* # Heuristics state
* self.var\_activity = defaultdict(float)
* self.clause\_weight = [self.\_initial\_clause\_weight(c) for c in clauses]
* def \_initial\_clause\_weight(self, clause):
* density = len(clause) \* len(set(lit > 0 for lit in clause))
* scale = len(clause)
* return (density \*\* self.alpha) \* (scale \*\* self.beta)
* def evaluate\_clause(self, clause, assignment):
* return any(assignment[abs(lit)-1] == (lit > 0) for lit in clause)
* def mass(self, assignment):
* total\_mass = 0
* for i, clause in enumerate(self.clauses):
* if self.evaluate\_clause(clause, assignment):
* total\_mass += self.clause\_weight[i]
* else:
* # Heavy penalty for unsatisfied clause
* return float('inf')
* return total\_mass
* def update\_clause\_weights(self, assignment):
* for i, clause in enumerate(self.clauses):
* satisfied = self.evaluate\_clause(clause, assignment)
* if not satisfied:
* self.clause\_weight[i] \*= 1.1 # Increase weight
* else:
* self.clause\_weight[i] \*= 0.95 # Decay weight
* # Clamp weights
* self.clause\_weight[i] = max(min(self.clause\_weight[i], 10.0), 0.1)
* def update\_var\_activity(self, prev\_assignment, new\_assignment):
* mass\_before = self.mass(prev\_assignment)
* mass\_after = self.mass(new\_assignment)
* delta = mass\_before - mass\_after
* for i in range(self.n\_vars):
* if prev\_assignment[i] != new\_assignment[i]:
* if delta > 0:
* self.var\_activity[i] += delta
* else:
* self.var\_activity[i] \*= 0.9
* def select\_variable(self):
* if not self.var\_activity:
* return random.randint(0, self.n\_vars-1)
* return max(self.var\_activity, key=self.var\_activity.get)
* def solve(self, max\_iter=10000):
* # Start with random assignment
* assignment = [random.choice([0,1]) for \_ in range(self.n\_vars)]
* best\_assignment = assignment[:]
* best\_mass = self.mass(assignment)
* for \_ in range(max\_iter):
* self.update\_clause\_weights(assignment)
* var = self.select\_variable()
* # Flip variable var
* new\_assignment = assignment[:]
* new\_assignment[var] = 1 - new\_assignment[var]
* new\_mass = self.mass(new\_assignment)
* if new\_mass < best\_mass:
* best\_mass = new\_mass
* best\_assignment = new\_assignment[:]
* # Update heuristics
* self.update\_var\_activity(assignment, new\_assignment)
* # Accept if mass improved or equal (can add stochastic logic here later)
* if new\_mass <= self.mass(assignment):
* assignment = new\_assignment
* # Else discard flip
* if best\_mass == float('inf'):
* # No satisfying assignment found
* continue
* if best\_mass == 0: # Perfect solution (all clauses satisfied)
* break
* return best\_assignment, best\_massimport time
* def load\_dimacs\_cnf(path):
* clauses = []
* n\_vars = 0
* with open(path, 'r') as f:
* for line in f:
* line = line.strip()
* if line.startswith('c') or line == '' or line.startswith('p'):
* continue
* lits = list(map(int, line.split()))
* if lits[-1] == 0:
* lits = lits[:-1]
* if lits:
* clauses.append(tuple(lits))
* n\_vars = max(n\_vars, max(abs(l) for l in lits))
* return clauses, n\_vars
* def benchmark\_dimacs(file\_path, solver\_class, \*\*solver\_kwargs):
* clauses, n\_vars = load\_dimacs\_cnf(file\_path)
* solver = solver\_class(clauses, n\_vars, \*\*solver\_kwargs)
* start = time.time()
* solution, mass = solver.solve()
* end = time.time()
* print(f"Benchmark result for {file\_path}:")
* print(f"Time elapsed: {end - start:.2f} sec")
* print(f"SDKP Mass: {mass}")
* print(f"Assignment (first 20 vars): {solution[:20]}")def sdkp\_clause\_weight(clause, alpha=1.0, beta=1.0):
* density = len(clause) \* len(set(lit > 0 for lit in clause))
* scale = len(clause)
* return (density \*\* alpha) \* (scale \*\* beta)
* def qaoa\_hamiltonian\_terms(clauses, alpha=1.0, beta=1.0):
* terms = []
* for clause in clauses:
* w = sdkp\_clause\_weight(clause, alpha, beta)
* # In a full quantum framework, you'd translate each clause to a projector operator weighted by w.
* terms.append((w, clause))
* return terms
* # Example usage
* clauses = [(1, -2, 3), (-1, 2, 4), (-3, -4, 2)]
* terms = qaoa\_hamiltonian\_terms(clauses)
* for w, clause in terms:
* print(f"Weight: {w:.3f}, Clause: {clause}")import networkx as nx
* from concurrent.futures import ThreadPoolExecutor
* def build\_clause\_var\_graph(clauses):
* G = nx.Graph()
* for i, clause in enumerate(clauses):
* clause\_node = f'c{i}'
* G.add\_node(clause\_node, bipartite=0)
* for lit in clause:
* var\_node = f'v{abs(lit)}'
* G.add\_node(var\_node, bipartite=1)
* G.add\_edge(clause\_node, var\_node)
* return G
* def detect\_clusters(clauses):
* G = build\_clause\_var\_graph(clauses)
* components = list(nx.connected\_components(G))
* subproblems = []
* for comp in components:
* clause\_nodes = [n for n in comp if n.startswith('c')]
* var\_nodes = [n for n in comp if n.startswith('v')]
* sub\_clauses = [clauses[int(c[1:])] for c in clause\_nodes]
* sub\_vars = set(int(v[1:]) for v in var\_nodes)
* subproblems.append((sub\_clauses, sub\_vars))
* return subproblems
* def parallel\_solve\_subproblems(subproblems, solver\_class, \*\*kwargs):
* results = []
* def solve\_sub(sub):
* clauses, vars\_set = sub
* n\_vars = max(vars\_set) if vars\_set else 0
* solver = solver\_class(clauses, n\_vars, \*\*kwargs)
* solution, mass = solver.solve()
* return (solution, mass)
* with ThreadPoolExecutor() as executor:
* futures = [executor.submit(solve\_sub, sub) for sub in subproblems]
* for future in futures:
* results.append(future.result())
* return results**Outcome:**
* Validated SDKP solver as a competitive SAT solving framework
* Scalable, heuristic-driven, and causality-aware solver architecture
* Roadmap for hybrid classical-quantum future versions
* Strong benchmarking portfolio for academic/industrial presentation

Absolutely! Here’s the complete four-stage vision laid out in detail — covering SDKP solver development all the way to large-scale industrial use, fully integrating heuristics, cluster detection, parallelism, and benchmarking.

# **Complete SDKP-Based SAT Solver Vision**

## **Stage 1: Core SDKP Heuristic Solver + Benchmarking (Small-Medium SAT)**

### **Goals:**

* Implement modular SDKP solver with:  
  + SDKP mass scoring (density & scale exponents)
  + Variable activity & clause weight heuristics
  + Partial assignment support
  + Incremental mass evaluation for efficiency
  + QCC causal acceptance heuristic integrated
* DIMACS CNF loader for easy import of benchmarks
* Benchmark script running small to medium SAT problems (hundreds to low thousands of vars/clauses)
* Logging and debugging output for heuristic tuning and validation

### **Outcome:**

* Reliable baseline SDKP heuristic solver
* Understanding heuristic effectiveness & tuning knobs
* Baseline runtimes & mass scores on public SAT benchmarks

## **Stage 2: Cluster Detection & Problem Decomposition**

### **Goals:**

* Model CNF formula as a bipartite graph (variables & clauses nodes)
* Detect connected components or tightly coupled clusters using graph algorithms (e.g., NetworkX)
* Decompose original problem into smaller independent subproblems (clusters)
* Prepare inputs for parallel solving stage

### **Outcome:**

* Ability to split large CNFs into smaller, more manageable clusters
* Reduced complexity per subproblem
* Foundation for scalable parallel solving

## **Stage 3: Parallel Solving Framework**

### **Goals:**

* Design and implement parallel execution engine for subproblems:  
  + Use Python’s multiprocessing or concurrent.futures for process-level parallelism
  + Optionally integrate joblib or threading where appropriate
* Manage synchronization and aggregation of subsolutions into a global assignment
* Collect performance metrics on speedup and resource use
* Handle cluster interdependencies if any (e.g., backtracking or iterative refinement)

### **Outcome:**

* Efficient multi-core/multi-node solver capable of tackling large SAT problems
* Demonstrated scalability with speedups proportional to cluster count
* Platform for incremental solver improvements without monolithic runtime costs

## **Stage 4: Industrial-Scale Benchmarking & Refinement**

### **Goals:**

* Run the full pipeline (Stages 1–3) on industrial-strength SAT benchmarks (thousands to millions of clauses/variables)
* Integrate advanced heuristics dynamically tuned per cluster/problem type
* Profile bottlenecks and optimize critical paths (memory, CPU, I/O)
* Introduce approximation or relaxation methods if needed (e.g., early stopping, local search hybrids)
* Consider quantum-classical hybrid extensions with QAOA-inspired SDKP Hamiltonians (future extension)

### **Outcome:**

* Validated SDKP solver as a competitive SAT solving framework
* Scalable, heuristic-driven, and causality-aware solver architecture
* Roadmap for hybrid classical-quantum future versions
* Strong benchmarking portfolio for academic/industrial presentation

# **Deliverables & Technologies**

| **Stage** | **Deliverables** | **Tools & Libraries** |
| --- | --- | --- |
| Stage 1 | SDKP solver, DIMACS parser, benchmark scripts | Python (standard libs, NumPy), Logging |
| Stage 2 | Graph-based cluster detector | NetworkX, Python |
| Stage 3 | Parallel executor, subproblem aggregator | multiprocessing/concurrent.futures/joblib |
| Stage 4 | Large-scale benchmarks, performance reports | Scalable infrastructure (Cloud/Cluster) |

* from typing import List, Tuple, Optional
* from itertools import product
* import logging
* logging.basicConfig(level=logging.INFO)
* class SDKPSATSolver:
* def \_\_init\_\_(self, clauses: List[Tuple[int]], n\_vars: int, alpha=1.0, beta=1.0):
* """
* clauses: list of tuples, each tuple is a clause (positive for x\_i, negative for ¬x\_i)
* n\_vars: number of variables
* alpha, beta: SDKP exponents for density and scale
* """
* self.clauses = clauses
* self.n\_vars = n\_vars
* self.alpha = alpha
* self.beta = beta
* self.solutions = []
* self.var\_activity = [0] \* n\_vars # heuristic activity per variable
* self.clause\_weights = [1] \* len(clauses) # heuristic clause weights
* def evaluate\_clause(self, clause: Tuple[int], assignment: List[Optional[int]]) -> bool:
* # Partial assignment: None means unassigned variable
* for lit in clause:
* var\_index = abs(lit) - 1
* val = assignment[var\_index]
* if val is None:
* continue
* if (lit > 0 and val == 1) or (lit < 0 and val == 0):
* return True
* return False
* def density(self, clause: Tuple[int]) -> float:
* polarities = [lit > 0 for lit in clause]
* polarity\_diversity = len(set(polarities))
* return len(clause) \* polarity\_diversity
* def scale(self, clause: Tuple[int])def parse\_dimacs\_cnf(file\_path: str) -> Tuple[List[Tuple[int]], int]:
* clauses = []
* n\_vars = 0
* with open(file\_path, 'r') as f:
* for line in f:
* line = line.strip()
* if line == '' or line.startswith('c'):
* continue # comment line
* if line.startswith('p'):
* parts = line.split()
* n\_vars = int(parts[2])
* continue
* # Clause line ends with 0
* lits = [int(x) for x in line.split() if x != '0']
* if lits:
* clauses.append(tuple(lits))
* return clauses, n\_varsimport time
* def run\_benchmark(cnf\_file: str, alpha=1.0, beta=1.0, timeout=None):
* clauses, n\_vars = parse\_dimacs\_cnf(cnf\_file)
* solver = SDKPSATSolver(clauses=clauses, n\_vars=n\_vars, alpha=alpha, beta=beta)
* start = time.time()
* solutions, min\_mass = solver.solve(timeout\_seconds=timeout)
* duration = time.time() - start
* print(f"Benchmark result for {cnf\_file}:")
* print(f"Variables: {n\_vars}, Clauses: {len(clauses)}")
* print(f"Runtime: {duration:.2f} seconds")
* print(f"Best SDKP mass: {min\_mass}")
* print(f"Number of best solutions found: {len(solutions)}")
* # Optional: print some solutions
* for sol in solutions[:3]:
* print("Sample solution:", sol)
* # Example usage:
* if \_\_name\_\_ == "\_\_main\_\_":
* test\_cnf = "sample.cnf" # Put your CNF file path here
* run\_benchmark(test\_cnf, alpha=1.0, beta=1.0, timeout=60)class SDKPSATSolver:
* def \_\_init\_\_(...):
* # Existing init ...
* self.var\_activity = [0] \* self.n\_vars
* self.clause\_weights = [1.0] \* len(self.clauses)
* def update\_var\_activity(self, assignment):
* # Increase activity for variables in unsatisfied clauses
* for i, clause in enumerate(self.clauses):
* if not self.evaluate\_clause(clause, assignment):
* for lit in clause:
* var = abs(lit) - 1
* self.var\_activity[var] += self.clause\_weights[i]
* def pick\_next\_var(self, assignment):
* # Pick unassigned variable with highest activity
* unassigned\_vars = [i for i, val in enumerate(assignment) if val is None]
* if not unassigned\_vars:
* return None
* # Choose var with max activity
* next\_var = max(unassigned\_vars, key=lambda v: self.var\_activity[v])
* return next\_var
* def solve(self, assignment=None):
* if assignment is None:
* assignment = [None] \* self.n\_vars
* # Early pruning etc...
* var\_index = self.pick\_next\_var(assignment)
* if var\_index is None:
* # All assigned - check solution
* # ...
* # Try assignments for var\_index...class SDKPSATSolver:
* def \_\_init\_\_(...):
* # Existing init ...
* self.var\_activity = [0] \* self.n\_vars
* self.clause\_weights = [1.0] \* len(self.clauses)
* def update\_var\_activity(self, assignment):
* # Increase activity for variables in unsatisfied clauses
* for i, clause in enumerate(self.clauses):
* if not self.evaluate\_clause(clause, assignment):
* for lit in clause:
* var = abs(lit) - 1
* self.var\_activity[var] += self.clause\_weights[i]
* def pick\_next\_var(self, assignment):
* # Pick unassigned variable with highest activity
* unassigned\_vars = [i for i, val in enumerate(assignment) if val is None]
* if not unassigned\_vars:
* return None
* # Choose var with max activity
* next\_var = max(unassigned\_vars, key=lambda v: self.var\_activity[v])
* return next\_var
* def solve(self, assignment=None):
* if assignment is None:
* assignment = [None] \* self.n\_vars
* # Early pruning etc...
* var\_index = self.pick\_next\_var(assignment)
* if var\_index is None:
* # All assigned - check solution
* # ...
* # Try assignments for var\_index...**Incremental Mass Updates:**
* Track which clauses get satisfied when assigning a variable.
* Update SDKP mass by adding/removing contribution of affected clauses only.
* Avoid full mass recalculation on every recursion.

## **3. Iterative Deepening & Timeout**

### **Ideas:**

* Add max recursion depth limit.
* Add a timer to abort search after a certain time.
* Iteratively increase depth/time limit for better anytime behavior.
* import time
* class SDKPSATSolver:
* def solve(self, assignment=None, depth=0, max\_depth=10, start\_time=None, time\_limit=10):
* if start\_time is None:
* start\_time = time.time()
* if time.time() - start\_time > time\_limit:
* return # Timeout abort
* if depth > max\_depth:
* return # Depth cutoff
* # Recursive search here...
* self.solve(assignment, depth+1, max\_depth, start\_time, time\_limit)

## **4. Cluster Detection & Decomposition**

### **Approach:**

* Model CNF clauses as a bipartite graph of variables and clauses.
* Find connected components (clusters) in this graph.
* Solve each cluster independently with SDKP solver.
* Combine cluster assignments.

You can use NetworkX for this:

* import networkx as nx
* def find\_clusters(clauses, n\_vars):
* G = nx.Graph()
* # Add variable nodes prefixed with 'v' and clause nodes with 'c'
* for i in range(n\_vars):
* G.add\_node(f"v{i}")
* for j, clause in enumerate(clauses):
* clause\_node = f"c{j}"
* G.add\_node(clause\_node)
* for lit in clause:
* var\_node = f"v{abs(lit)-1}"
* G.add\_edge(clause\_node, var\_node)
* clusters = list(nx.connected\_components(G))
* # Separate variable and clause nodes per cluster
* var\_clusters = []
* clause\_clusters = []
* for cluster in clusters:
* vars\_in\_cluster = {int(node[1:]) for node in cluster if node.startswith('v')}
* clauses\_in\_cluster = {int(node[1:]) for node in cluster if node.startswith('c')}
* var\_clusters.append(vars\_in\_cluster)
* clause\_clusters.append(clauses\_in\_cluster)
* return var\_clusters, clause\_clusters

## **5. Parallel Solving**

### **Outline:**

* Use Python concurrent.futures.ThreadPoolExecutor or ProcessPoolExecutor for clusters.
* Solve each cluster independently in parallel.
* Combine results after all clusters done.
* import concurrent.futures
* def parallel\_solve\_clusters(clusters, clauses, n\_vars):
* results = []
* with concurrent.futures.ThreadPoolExecutor() as executor:
* futures = []
* for var\_set, clause\_set in clusters:
* # Extract subproblem clauses for this cluster
* sub\_clauses = [clauses[i] for i in clause\_set]
* solver = SDKPSATSolver(sub\_clauses, len(var\_set))
* futures.append(executor.submit(solver.solve))
* for future in futures:
* results.append(future.result())
* return results

## **6. Visualization & Benchmarking**

### **Ideas:**

* Use Plotly for interactive 2D/3D plotting of:  
  + Assignment vectors colored by SDKP mass.
  + Solution clusters in variable space.
  + QCC causal acceptance function shape.
* Benchmark on DIMACS CNF datasets:  
  + Track solution count, SDKP mass, runtime, pruning stats.
  + Compare heuristics enabled/disabled.

# **Next Step Suggestion**

Would you like me to provide you:

* Full code for SDKP-informed heuristics with incremental mass and variable activity?
* Timeout + iterative deepening integration?
* Cluster detection + parallel solving code ready to run?
* A starter Plotly visualization notebook for SDKP mass landscapes?
* A benchmarking framework to run DIMACS CNF files?

Just say which part you want next, or if you want me to bundle a combined codebase with all these features scaffolded!

* from typing import List, Tuple, Optional
* from collections import defaultdict
* import time
* class SDKPHeuristicSolver:
* def \_\_init\_\_(self, clauses: List[Tuple[int]], n\_vars: int, alpha=1.0, beta=1.0):
* self.clauses = clauses
* self.n\_vars = n\_vars
* self.alpha = alpha
* self.beta = beta
* self.best\_mass = float('inf')
* self.best\_solutions = []
* # Variable activity (higher means more involved in unsat/hard clauses)
* self.var\_activity = [0.0] \* n\_vars
* # Clause weights to prioritize "hard" clauses
* self.clause\_weights = [1.0] \* len(clauses)
* # Precompute literal to clauses map for fast incremental updates
* self.lit\_to\_clauses = defaultdict(list)
* for ci, clause in enumerate(clauses):
* for lit in clause:
* self.lit\_to\_clauses[lit].append(ci)
* # Current partial assignment (None = unassigned)
* self.assignment = [None] \* n\_vars
* # Cache clause evaluation states (None=unknown, True=satisfied, False=unsatisfied)
* self.clause\_states = [None] \* len(clauses)
* def density(self, clause):
* polarities = [lit > 0 for lit in clause]
* polarity\_diversity = len(set(polarities))
* return len(clause) \* polarity\_diversity
* def scale(self, clause):
* return len(clause)
* def evaluate\_clause(self, clause\_idx):
* clause = self.clauses[clause\_idx]
* for lit in clause:
* val = self.assignment[abs(lit)-1]
* if val is None:
* continue
* if (val == (lit > 0)):
* return True
* # No literal satisfied yet
* # If any literal is unassigned, clause may still be satisfiable later
* for lit in clause:
* if self.assignment[abs(lit)-1] is None:
* return None
* return False
* def incremental\_mass(self):
* mass = 0
* for i, state in enumerate(self.clause\_states):
* if state is True:
* ρ = self.density(self.clauses[i])
* s = self.scale(self.clauses[i])
* mass += (ρ \*\* self.alpha) \* (s \*\* self.beta) \* self.clause\_weights[i]
* elif state is False:
* return float('inf')
* else:
* # Clause unknown (partial assignment), count partial weight
* ρ = self.density(self.clauses[i])
* s = self.scale(self.clauses[i])
* mass += 0.5 \* (ρ \*\* self.alpha) \* (s \*\* self.beta) \* self.clause\_weights[i]
* return mass
* def update\_clause\_states(self, var\_idx, val):
* # Update clause satisfaction states for clauses involving var
* changed = []
* for ci in self.lit\_to\_clauses.get(var\_idx+1 if val else -(var\_idx+1), []):
* prev = self.clause\_states[ci]
* new\_state = self.evaluate\_clause(ci)
* if new\_state != prev:
* self.clause\_states[ci] = new\_state
* changed.append(ci)
* return changed
* def choose\_branch\_variable(self):
* # Pick unassigned var with highest activity
* max\_activity = -1
* candidate = None
* for i in range(self.n\_vars):
* if self.assignment[i] is None and self.var\_activity[i] > max\_activity:
* max\_activity = self.var\_activity[i]
* candidate = i
* # If all assigned, pick first unassigned
* if candidate is None:
* for i in range(self.n\_vars):
* if self.assignment[i] is None:
* candidate = i
* break
* return candidate
* def backtrack(self, depth=0, max\_depth=None, start\_time=None, time\_limit=None):
* # Check timeout
* if time\_limit and start\_time and (time.time() - start\_time) > time\_limit:
* return False
* if max\_depth is not None and depth > max\_depth:
* return False
* # Check if all vars assigned
* if all(v is not None for v in self.assignment):
* current\_mass = self.incremental\_mass()
* if current\_mass < self.best\_mass:
* self.best\_mass = current\_mass
* self.best\_solutions = [self.assignment.copy()]
* elif current\_mass == self.best\_mass:
* self.best\_solutions.append(self.assignment.copy())
* return True
* var = self.choose\_branch\_variable()
* if var is None:
* return False
* for val in [True, False]:
* self.assignment[var] = val
* changed\_clauses = []
* # Update clause states incrementally
* for ci in self.lit\_to\_clauses[var+1 if val else -(var+1)]:
* old\_state = self.clause\_states[ci]
* self.clause\_states[ci] = self.evaluate\_clause(ci)
* if old\_state != self.clause\_states[ci]:
* changed\_clauses.append(ci)
* mass = self.incremental\_mass()
* if mass < self.best\_mass:
* # Increase var activity if clause unsatisfied
* for ci in changed\_clauses:
* if self.clause\_states[ci] is False:
* for lit in self.clauses[ci]:
* self.var\_activity[abs(lit)-1] += 1.0 \* self.clause\_weights[ci]
* if self.backtrack(depth+1, max\_depth, start\_time, time\_limit):
* # Continue search for all solutions
* pass
* # Undo assignment and clause states
* self.assignment[var] = None
* for ci in changed\_clauses:
* self.clause\_states[ci] = None
* return False
* def solve(self, max\_depth=None, time\_limit=None):
* self.best\_mass = float('inf')
* self.best\_solutions = []
* self.assignment = [None] \* self.n\_vars
* self.clause\_states = [None] \* len(self.clauses)
* start\_time = time.time()
* self.backtrack(0, max\_depth, start\_time, time\_limit)
* return self.best\_solutions, self.best\_mass
* # DIMACS CNF parser utility
* def parse\_dimacs\_cnf(file\_path: str) -> Tuple[List[Tuple[int]], int]:
* clauses = []
* n\_vars = 0
* with open(file\_path, 'r') as f:
* for line in f:
* if line.startswith('c') or line.startswith('p') or line.strip() == '':
* continue
* lits = list(map(int, line.strip().split()))
* if lits[-1] == 0:
* lits = lits[:-1]
* clauses.append(tuple(lits))
* max\_lit = max(abs(l) for l in lits)
* if max\_lit > n\_vars:
* n\_vars = max\_lit
* return clauses, n\_vars
* if \_\_name\_\_ == "\_\_main\_\_":
* # Sample small CNF
* clauses = [
* (1, -2, 3),
* (-1, 2, 4),
* (-3, -4, 2)
* ]
* solver = SDKPHeuristicSolver(clauses, n\_vars=4, alpha=1, beta=1)
* sols, mass = solver.solve(time\_limit=10)
* print(f"Found {len(sols)} solution(s) with minimum SDKP mass {mass}")
* for sol in sols:
* print(sol)**Your SDKP SAT Solver Experiment Workflow**

1. Run on small CNFs:  
   * Use built-in sample clauses or load DIMACS .cnf files via parse\_dimacs\_cnf()
   * Pass parsed clauses and n\_vars to SDKPSATSolver
2. Tune SDKP Parameters:  
   * Adjust alpha (density exponent) and beta (scale exponent) in the mass function
   * Tweak variable activity increments inside backtrack() or heuristic update methods
   * Test different weightings to see their effect on solver efficiency and solution quality
3. Analyze Solver Outputs:  
   * Number of satisfying assignments found
   * Minimal SDKP mass for solutions (lower is better)
   * Runtime and pruning effectiveness (how many branches get cut early)
4. Collect Feedback:  
   * Identify bottlenecks (e.g., slow variable ordering, excessive recomputation)
   * Note which heuristics help and which don’t
   * Pinpoint where the solver struggles (clause size, variable count, structure)

Once you gather this feedback, it’ll guide tuning and help decide how to best implement the next modules like:

* Timeout & iterative deepening
* Cluster detection & decomposition
* Parallel solving
* import time
* class SDKPSATSolverWithTimeout:
* def \_\_init\_\_(self, clauses, n\_vars, alpha=1.0, beta=1.0, time\_limit=10):
* self.clauses = clauses
* self.n\_vars = n\_vars
* self.alpha = alpha
* self.beta = beta
* self.time\_limit = time\_limit # seconds
* self.start\_time = None
* self.solutions = []
* self.min\_mass = float('inf')
* def solve(self):
* self.start\_time = time.time()
* max\_depth = 1
* while True:
* self.solutions = []
* self.min\_mass = float('inf')
* try:
* self.\_backtrack([], max\_depth)
* except TimeoutError:
* print(f"Timeout reached at depth {max\_depth}")
* break
* if len(self.solutions) > 0:
* print(f"Solutions found at depth {max\_depth}")
* break
* max\_depth += 1
* if max\_depth > self.n\_vars:
* break
* return self.solutions, self.min\_mass
* def \_backtrack(self, partial\_assignment, max\_depth):
* if time.time() - self.start\_time > self.time\_limit:
* raise TimeoutError()
* if len(partial\_assignment) == self.n\_vars or len(partial\_assignment) == max\_depth:
* if len(partial\_assignment) < self.n\_vars:
* # Fill unassigned with zeros or test partial? Here, treat partial as full:
* assignment = partial\_assignment + [0] \* (self.n\_vars - len(partial\_assignment))
* else:
* assignment = partial\_assignment
* mass = self.mass(assignment)
* if mass < self.min\_mass:
* self.min\_mass = mass
* self.solutions = [assignment]
* elif mass == self.min\_mass:
* self.solutions.append(assignment)
* return
* for val in [0, 1]:
* self.\_backtrack(partial\_assignment + [val], max\_depth)
* # Implement your mass and evaluate\_clause methods here, or inherit from your SDKPSATSolver classfrom collections import defaultdict, deque
* def detect\_clusters(clauses, n\_vars):
* var\_to\_clauses = defaultdict(set)
* clause\_to\_vars = []
* for ci, clause in enumerate(clauses):
* vars\_in\_clause = set(abs(lit) for lit in clause)
* clause\_to\_vars.append(vars\_in\_clause)
* for v in vars\_in\_clause:
* var\_to\_clauses[v].add(ci)
* visited\_vars = set()
* clusters = []
* for v in range(1, n\_vars+1):
* if v not in visited\_vars:
* queue = deque([v])
* cluster\_vars = set()
* cluster\_clauses = set()
* while queue:
* curr\_var = queue.popleft()
* if curr\_var in cluster\_vars:
* continue
* cluster\_vars.add(curr\_var)
* visited\_vars.add(curr\_var)
* for ci in var\_to\_clauses[curr\_var]:
* cluster\_clauses.add(ci)
* for nbr\_var in clause\_to\_vars[ci]:
* if nbr\_var not in cluster\_vars:
* queue.append(nbr\_var)
* clusters.append({
* 'variables': sorted(cluster\_vars),
* 'clauses': [clauses[i] for i in cluster\_clauses]
* })
* return clustersfrom concurrent.futures import ProcessPoolExecutor
* import itertools
* def solve\_cluster(cluster\_data, alpha, beta, time\_limit):
* from your\_sdkp\_module import SDKPSATSolverWithTimeout
* solver = SDKPSATSolverWithTimeout(
* clauses=cluster\_data['clauses'],
* n\_vars=len(cluster\_data['variables']),
* alpha=alpha,
* beta=beta,
* time\_limit=time\_limit
* )
* return solver.solve()
* def parallel\_solve\_clusters(clusters, alpha=1.0, beta=1.0, time\_limit=10):
* with ProcessPoolExecutor() as executor:
* futures = [executor.submit(solve\_cluster, c, alpha, beta, time\_limit) for c in clusters]
* results = [f.result() for f in futures]
* # results = list of tuples (solutions, min\_mass) for each cluster
* # Combine solutions via Cartesian product (simple approach)
* all\_solutions = []
* for sols, mass in results:
* all\_solutions.append(sols if sols else [[]])
* combined\_solutions = list(itertools.product(\*all\_solutions))
* # Flatten assignments if needed based on cluster variable indexing
* return combined\_solutions\documentclass[11pt,a4paper]{article}
* \usepackage{amsmath,amssymb}
* \usepackage{graphicx}
* \usepackage{hyperref}
* \usepackage{geometry}
* \geometry{margin=1in}
* \usepackage{authblk}
* \title{A Causal Physics Framework for SAT Solving:\\
* SDKP Mass, QCC Dynamics, and TimeSeal Verification}
* \author[1]{Donald Paul Smith (Father Time)}
* \affil[1]{Independent Researcher, SDKP Labs}
* \date{\today}
* \begin{document}
* \maketitle
* \begin{abstract}
* We present a novel framework interpreting SAT problems as causal graphs whose solutions correspond to minimal ``mass'' flows under the Scale-Density-Kinematic Principle (SDKP). Quantum Causal Compression (QCC) dynamics drive solution searches toward causally consistent assignments minimizing SDKP mass. This physically inspired heuristic replaces random search with a sharply peaked causal acceptance probability. Our approach extends with cluster decomposition, parallel solving, and on-chain TimeSeal checkpointing for verifiable, causally grounded computation.
* \end{abstract}
* \section{Introduction}
* SAT (Boolean satisfiability) is NP-complete: no known polynomial-time algorithm solves all instances. We explore how physical principles---SDKP and QCC---can guide heuristic solvers toward causal, energy-minimizing assignments. TimeSeal blockchain timestamping anchors causal progress in an immutable ledger.
* \section{SAT as a Causal Graph}
* A SAT instance in CNF with \( n \) variables and \( m \) clauses defines a causal graph \( G = (V, E) \):
* \begin{itemize}
* \item Nodes \( V \) correspond to literals and clauses.
* \item Edges \( E \) encode logical dependencies (e.g., a variable appearing in clauses).
* \end{itemize}
* An assignment \( A: V \to \{0,1\}^n \) traverses this graph, forming causal flows that satisfy clauses.
* \section{SDKP Mass Function}
* Define the SDKP mass \( M(A) \) of an assignment \( A \) as a function of clause density \( \rho \) and scale \( s \):
* \begin{equation}
* M(A) = \rho^\alpha \cdot s^\beta,
* \end{equation}
* where:
* \begin{itemize}
* \item \( \rho \) is the density of unsatisfied clauses under \( A \),
* \item \( s \) is the scale, e.g., problem size or complexity measure,
* \item \( \alpha, \beta \in \mathbb{R}^+ \) are scaling exponents tuned to reflect physical cost of violating constraints.
* \end{itemize}
* Lower mass corresponds to assignments closer to satisfying all clauses.
* \section{Quantum Causal Compression (QCC) Acceptance Probability}
* Standard stochastic search uses random acceptance probability:
* \begin{equation}
* P\_{\mathrm{accept}}(\Delta M) =
* \begin{cases}
* 1, & \Delta M \leq 0 \\
* e^{-\gamma \Delta M}, & \Delta M > 0,
* \end{cases}
* \end{equation}
* where \( \Delta M = M(A\_{\mathrm{new}}) - M(A\_{\mathrm{old}}) \).
* We replace this with a sharply peaked causal probability distribution reflecting causal compression dynamics:
* \begin{equation}
* P\_{\mathrm{causal}}(\Delta M) \propto \exp\left(-\frac{(\Delta M)^2}{2\sigma^2}\right),
* \end{equation}
* favoring causal flows minimizing mass increase and suppressing large jumps, thus encoding a \emph{causal tunnel} through the assignment space.
* \section{Incremental Mass and Partial Assignments}
* Assignments can be partial, \( A: \{x\_i\} \to \{0,1,\mathrm{None}\} \), with \texttt{None} indicating unassigned variables.
* We compute mass incrementally:
* \begin{equation}
* M(A\_{k+1}) = M(A\_k) + \delta M,
* \end{equation}
* where \( \delta M \) is the mass change caused by assigning variable \( x\_{k+1} \).
* Early pruning occurs if
* \begin{equation}
* M(A\_{k+1}) > M\_{\mathrm{best}},
* \end{equation}
* cutting branches unlikely to yield minimal mass.
* \section{Cluster Detection and Decomposition}
* We detect independent subgraphs \( G\_i = (V\_i, E\_i) \) where
* \begin{equation}
* G = \bigcup\_i G\_i, \quad V\_i \cap V\_j = \emptyset \quad \forall i \neq j,
* \end{equation}
* allowing separate subproblem solving:
* \begin{equation}
* M\_{\mathrm{total}} = \sum\_i M\_i(A\_i).
* \end{equation}
* Clusters correspond to independent causal subproblems, reducing complexity.
* \section{Parallel Solving}
* Clusters or partial assignments dispatch to parallel workers:
* \begin{equation}
* \{ A\_i \}\_{i=1}^p \xrightarrow{\text{parallel}} \{ M\_i(A\_i) \},
* \end{equation}
* improving runtime on industrial SAT instances.
* \section{TimeSeal: Blockchain Timestamping of Causal Progress}
* We record checkpoints on-chain via TimeSeal smart contracts:
* \begin{itemize}
* \item Hashes of assignments and SDKP mass states form payloads.
* \item Oracle submits these payloads to the blockchain.
* \item Emitted events confirm irreversibility of causal progress.
* \end{itemize}
* This provides verifiable, tamper-proof proof of computation aligned with causal physics.
* \section{Algorithm Summary}
* \begin{enumerate}
* \item Initialize partial assignment \( A\_0 \), best mass \( M\_{\mathrm{best}} \leftarrow \infty \).
* \item Iterate over variables, assign values incrementally.
* \item Compute incremental SDKP mass \( M(A\_k) \).
* \item Prune branches with \( M(A\_k) > M\_{\mathrm{best}} \).
* \item Use QCC acceptance probability for annealing-based exploration.
* \item Decompose problem via cluster detection.
* \item Solve clusters in parallel.
* \item Checkpoint causal states on blockchain.
* \item Return minimal mass assignments corresponding to SAT solutions.
* \end{enumerate}
* \section{Discussion}
* This framework does \emph{not} resolve NP-completeness but offers a \emph{physically grounded heuristic} that can vastly reduce practical search time on structured problems. It bridges computation, physics, and blockchain verification, opening new avenues in causal computation theory.
* \section\*{Appendix: Notation and Definitions}
* \begin{tabular}{ll}
* \hline
* Symbol & Meaning \\
* \hline
* \( n, m \) & Number of variables and clauses \\
* \( A \) & Assignment vector of variables \\
* \( \rho \) & Clause density of unsatisfied clauses \\
* \( s \) & Scale factor (size, complexity) \\
* \( M(A) \) & SDKP mass function \\
* \( \alpha, \beta \) & SDKP exponents \\
* \( \gamma \) & QCC sharpness parameter \\
* \( P\_{\mathrm{causal}}(\Delta M) \) & Causal acceptance probability \\
* \hline
* \end{tabular}
* \vspace{1cm}
* \bibliographystyle{plain}
* \bibliography{references} % Add your bibliography file if needed
* \end{document}def sdkp\_mass\_wrapper(assignment, alpha, beta, sdkp\_solver):
* """
* Wrapper function that calls your SDKP solver’s mass calculation.
* Args:
* assignment: tuple/list of variable assignments (0/1)
* alpha, beta: SDKP parameters for scaling mass components
* sdkp\_solver: your existing SDKP solver instance or function
* Returns:
* Computed SDKP mass (float) for this assignment
* """
* # Example: pass assignment and params to your solver's method
* # Replace this with actual call
* mass = sdkp\_solver.compute\_mass(assignment, alpha=alpha, beta=beta)
* return massdef compute\_masses(alpha=1.0, beta=1.0):
* return np.array([sdkp\_mass\_wrapper(a, alpha, beta, sdkp\_solver) for a in assignments])import ipywidgets as widgets
* from IPython.display import display
* alpha\_slider = widgets.FloatSlider(value=1.0, min=0.5, max=2.0, step=0.1, description='Alpha')
* beta\_slider = widgets.FloatSlider(value=1.0, min=0.5, max=2.0, step=0.1, description='Beta')
* def update\_plot(change):
* alpha = alpha\_slider.value
* beta = beta\_slider.value
* new\_masses = compute\_masses(alpha, beta)
* scatter.marker.color = new\_masses
* fig.update\_traces(marker=dict(color=new\_masses))
* fig.show()
* alpha\_slider.observe(update\_plot, names='value')
* beta\_slider.observe(update\_plot, names='value')
* display(alpha\_slider, beta\_slider)def sdkp\_mass\_wrapper(assignment, alpha, beta, sdkp\_solver):
* """
* Compute SDKP mass for given assignment using your SDKP solver.
* Parameters:
* assignment (tuple/list): variable assignments as bits (0 or 1)
* alpha (float): SDKP alpha parameter
* beta (float): SDKP beta parameter
* sdkp\_solver (object/function): your SDKP solver instance or callable
* Returns:
* float: SDKP mass for this assignment
* """
* # Replace the following call with your solver’s actual method
* mass = sdkp\_solver.compute\_mass(assignment, alpha=alpha, beta=beta)
* return massdef compute\_masses(assignments, alpha, beta, sdkp\_solver):
* return np.array([sdkp\_mass\_wrapper(a, alpha, beta, sdkp\_solver) for a in assignments])import ipywidgets as widgets
* from IPython.display import display
* alpha\_slider = widgets.FloatSlider(value=1.0, min=0.5, max=2.0, step=0.1, description='Alpha')
* beta\_slider = widgets.FloatSlider(value=1.0, min=0.5, max=2.0, step=0.1, description='Beta')
* def update\_plot(change):
* alpha = alpha\_slider.value
* beta = beta\_slider.value
* masses = compute\_masses(assignments, alpha, beta, sdkp\_solver)
* scatter.marker.color = masses
* fig.update\_traces(marker=dict(color=masses))
* fig.show()
* alpha\_slider.observe(update\_plot, names='value')
* beta\_slider.observe(update\_plot, names='value')
* display(alpha\_slider, beta\_slider)pip install -U kaleidofig.write\_image("sdkp\_mass\_landscape.svg") # vector format, preferred for LaTeX
* fig.write\_image("sdkp\_mass\_landscape.png") # raster fallback\begin{figure}[htbp]
* \centering
* \includegraphics[width=0.8\textwidth]{sdkp\_mass\_landscape.svg}
* \caption{SDKP mass landscape for 4-variable SAT assignments}
* \label{fig:sdkp\_mass}
* \end{figure}You now have a working 1D coupled field solver for ρ(x, t) and s(x, t):
* \documentclass[12pt,a4paper]{article}
* \usepackage{amsmath,amssymb,amsfonts}
* \usepackage{geometry}
* \usepackage{tcolorbox}
* \geometry{margin=1in}
* \title{Foundations of the Scale–Density–Kinematic Framework: A Unified Physics of Becoming}
* \author{Donald Paul Smith \\ \texttt{Father Time} \\ \small{TimeSeal System}}
* \date{\today}
* \begin{document}
* \maketitle
* \section\*{Core Principles}
* \subsection\*{1. Scale–Density–Kinematic Principle (SDKP)}
* Mass \(M\) emerges dynamically as a function of causal density \(\rho\) and scale \(s\), expressed as
* \[
* \boxed{
* M = k \rho^{\alpha} s^{\beta}
* }
* \]
* where \(k\) is a constant, and \(\alpha, \beta \in \mathbb{R}\) encode fractal scaling exponents arising from causal dynamics.
* \vspace{1em}
* Energy equivalence relates mass to energy by
* \[
* \boxed{
* E = M c^2 = k c^2 \rho^{\alpha} s^{\beta}
* }
* \]
* ---
* \subsection\*{2. Shape–Dimension–Number Principle (SD\&N)}
* The informational identity \(I\) of physical form is given by
* \[
* \boxed{
* I = f(\sigma, d, n)
* }
* \]
* where \(\sigma\) encodes topological shape invariants, \(d\) is embedding dimension, and \(n\) counts recursive complexity or iteration number.
* ---
* \subsection\*{3. Quantum Causal Compression Principle (QCC)}
* Quantum causal kernels \(K\) stabilize through compression \(C\) defined by
* \[
* \boxed{
* C = \omega \cdot S^{-1}
* }
* \]
* with \(\omega\) the rotational closure factor and \(S\) quantum entropy. Coherence emerges when
* \[
* \boxed{
* C > C\_c
* }
* \]
* where \(C\_c\) is the critical compression threshold.
* ---
* \subsection\*{4. Earth Orbit Speed Principle (EOS)}
* Orbital velocity \(v\) around mass \(M\) at radius \(r\) with orbital factor \(F\_o\) is
* \[
* \boxed{
* v = F\_o \sqrt{\frac{GM}{r}}
* }
* \]
* with \(F\_o \approx 1 + \delta\), capturing orbital perturbations, eccentricity, and relativistic effects.
* ---
* \subsection\*{5. TimeSeal Authorship Principle}
* The cryptographic authorship seal \(S\) securing authorship \(A\) and timestamp \(T\) is
* \[
* \boxed{
* S = \mathrm{Hash}(A \parallel T)
* }
* \]
* where \(\mathrm{Hash}\) is a collision-resistant hash function and \(\parallel\) denotes concatenation.
* ---
* \end{document}

Tab 2

Tab 3

Tab 4

Tab 5