Akademia Sztuki Kwantowej — Kwantowe Wyżarzanie

Materiały do 2-dniowego szkolenia z kwantowego wyżarzania (quantum annealing) i kombinatorycznych problemów optymalizacyjnych.

Zawartość kursu

```
Kwantowe_wyzarzanie_kombinatorycznych_problemow_optymalizacyjnych/
  Dzien_1/   - teoria (model Isinga, QUBO, twierdzenie adiabatyczne, D-Wave)
  Dzien_2/   - praktyka (SA, symulowana bifurkacja, branch & bound, GPU)
  pliki_pomocnicze/   - moduły Python + instancje testowe
```

Dzień 1 (teoria): 1. Przygotowanie środowiska 2. Klasyczny model Isinga 3. Model QUBO — kodowanie problemów 4. Przykłady: Max-Cut, kolorowanie grafu, TSP 5. Algorytm wyczerpującego przeszukiwania (brute force) 6. Przegląd algorytmów heurystycznych 7. Kwantowy model Isinga 8. Twierdzenie adiabatyczne i wyżarzanie kwantowe 9. Procesor D-Wave

Dzień 2 (praktyka): 1. Praca z wyżarzaczami kwantowymi (D-Wave Ocean SDK) 2. Symulowane wyżarzanie (SA) 3. Symulowana bifurkacja (SB) 4. Wyżarzanie równoległe 5. Heurystyczny branch & bound 6. Implementacja GPU (CuPy)

Szybki start

```bash
# 1. Sklonuj repozytorium
git clone https://github.com/iitis/AkademiaSztukiKwantowej.git
cd AkademiaSztukiKwantowej

# 2. Zainstaluj zależności do większości notebooków
pip install -r requirements.txt

# 3. Uruchom Jupyter
jupyter notebook
```

> Notebooki korzystające z D-Wave wymagają konta na https://cloud.dwavesys.com i skonfigurowanego tokenu (`dwave config create`).
>
> Notebook GPU (`Dzien_2/06_GPU.ipynb`) wymaga osobnej instalacji CuPy dopasowanej do lokalnej wersji CUDA.
>
> Notebooki z katalogu `Dzien_2/benchmarks/` wymagają dodatkowego środowiska eksperymentalnego (GPU, CuPy/PyTorch CUDA oraz zewnętrzne biblioteki takie jak `simulated_bifurcation`, `omnisolver`, `scikit-learn`) i nie są częścią domyślnego smoke testu.

Testy

```bash
pip install pytest
pytest tests/ -v
```

Testy weryfikują poprawność matematyczną funkcji konwertujących Ising ↔ QUBO oraz obliczania energii.

Struktura plików pomocniczych

Plik	Co robi
`funkcje_pomocnicze.py`	Wczytywanie instancji, konwersja Ising↔QUBO, obliczanie energii
`funkcje_pomocnicze_gpu.py`	Wersja GPU (CuPy) obliczania energii

Plik	Co robi
`generacja_instancji.py`	Generowanie losowych instancji Isinga na grafach Pegasus/Grid
`instancje/`	Gotowe instancje testowe (P2–P16, Grid5–Grid100, K8)

CI

GitHub Actions uruchamia automatycznie: - **testy jednostkowe** przy każdym push - **nieblokujący smoke test notebooków** przy push do `master` / `main` oraz przy PR do głównej gałęzi

Smoke test pomija notebooki wymagające zewnętrznego środowiska (D-Wave Cloud, GPU/CuPy oraz katalog `Dzien_2/benchmarks`), więc jego celem jest szybkie wykrywanie regresji uruchomieniowych w pozostałych materiałach.

Licencja

Apache 2.0 — szczegóły w pliku LICENSE.

00_Przygotowanie_srodowiska_pracy

May 26, 2026

1 Pozyskanie plików

Wejdź na https://github.com/iitis/AkademiaSztukiKwantowej - jest to repozytorium z tym szkoleniem. Systemy Linux z reguły mają docelowo zainstalowane Git. W pozostałych przypadkach trzeba go zainstalować ręcznie. Można to zrobić tutaj.

1. kopiuj URL repozytorium z GitHub (zakładka Code → HTTPS).
2. W terminalu przejdź do katalogu, gdzie chcesz je mieć i wykonaj:

git clone https://github.com/iitis/AkademiaSztukiKwantowej.git

3. Wejdź do katalogu repo:

cd AkademiaSztukiKwantowej

4. Przełącz się na gałąź ze szkoleniem

git checkout bg/quantum_annealing

5. Polecane jest utworzenie własnej gałęzi. Pozwoli to trzymać własne rozwiązania w repozytorium

git branch nazwa_galezi

git checkout nazwa_galezi

2 Konfiguracja głównego środowiska pracy

2.1 Niezbędne minimum

Zainstaluj język programowania Python (do ściągnięcia tutaj) w wersji 3.12. Następnie postępuj od kroku 5 w standardowej instalacji.

2.2 Standardowa instalacja

W trakcie szkoleniu będziemy głównie posługiwać się językiem programowania Python. W komputerach kwantowych często jest używany jako API. Zakładamy korzystanie z Anaconda - dystrybucji Pythona dedykowanej analizie danych i zastosowaniom naukowych. Do instalacji, aktualizacji i zarządzania środowiskami korzystać będziemy z menedżera pakietów Conda.

1. Zainstaluj conda

- Przejdź na oficjalną stronę Anaconda, gdzie możesz pobrać wybrany instalator (Miniconda, Anaconda Distribution lub Miniforge). Więcej informacji o każdym z tych instalatorów znajdziesz w dokumentacji instalacji conda. Po pobraniu, zainstaluj conda.

Kolejne kroki wykonujemy w terminalu systemowym / conda

2. Utwórz środowisko wirtualne.
   - W teminalu systemowym/anaconda prompt wpisz:

conda create –name szkolenia

3. Aktywuj środowisko:
   - W teminalu/anaconda prompt wpisz:

conda activate szkolenia

ilekroć będziemy przełączać się między środowiskami należy zdezaktywować bieżące środowisko `conda deactivate` i aktywować inne.

4. Zainstaluj Pythona w wymaganej wersji. Conda pozwala operować różnymi wersjami pythona dla każdego środowiska.
   - W teminalu/anaconda prompt wpisz:

conda install python=3.12

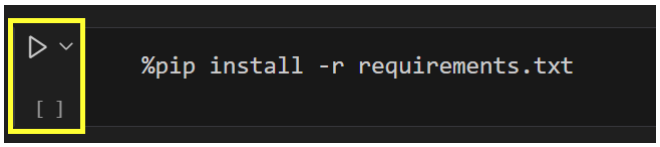
weryfikację instalacji można wykonać komendą `python --version`

5. Instalacja Ipykernel - backend dla notatnika Jupyter.

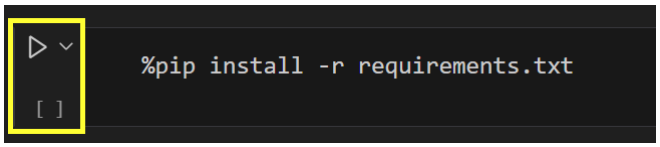
pip install ipykernel

Przechodzimy do narzędzia obsługującego notatniki Jupyter (np. może to być VS-Code).

6. Otwieramy folder z plikami szkoleniowymi. Wybieramy plik Setup.ipynb oraz ustawiamy jądro. W VSCode należy po otwarciu pliku wybrać (1) `Select Kernel`, a następnie na rozwijanej liście (2) wybrać utworzone wcześniej środowisko `szkolenia`



Tak skonfigurowane środowisko pozwoli uruchamiać kolejne komendy za pomocą wykonywalnych komórek notatnika. Aby uruchomić komórkę wystarczy kliknąć na ikonę ‘play’ obok komórki.



7. Instalacja paczek z pliku `requirements.txt`.

Uwaga! Jeśli z jakiegoś powodu plik znajduje się poza dostarczonym folderem będzie trzeba podać do niego pełną ścieżkę, zwróć uwagę na podwójne ukośniki.

"C:\\Users\\NAZWA_USERA\\Downloads\\requirements.txt"

[1]:
```
%pip install -r requirements.txt
```

```
DEPRECATION: Loading egg at
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-
packages/pyaml-25.1.0-py3.12.egg is deprecated. pip 24.3 will enforce this
behaviour change. A possible replacement is to use pip for package installation.
Discussion can be found at https://github.com/pypa/pip/issues/12330
Requirement already satisfied: matplotlib==3.10.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 1)) (3.10.0)
Requirement already satisfied: networkx==3.4.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 2)) (3.4.2)
Requirement already satisfied: numpy==2.0.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 3)) (2.0.2)
Requirement already satisfied: pandas==2.2.3 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 4)) (2.2.3)
Requirement already satisfied: tqdm==4.67.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 5)) (4.67.1)
Requirement already satisfied: dwave-ocean-sdk==8.2.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 6)) (8.2.0)
Requirement already satisfied: qutip==5.1.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 7)) (5.1.1)
Requirement already satisfied: pyqubo==1.5.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
-r requirements.txt (line 8)) (1.5.0)
Requirement already satisfied: contourpy>=1.0.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (4.55.7)
Requirement already satisfied: kiwisolver>=1.3.1 in
```

```
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (24.2)
Requirement already satisfied: pillow>=8 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
matplotlib==3.10.0->-r requirements.txt (line 1)) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pandas==2.2.3->-r requirements.txt (line 4)) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pandas==2.2.3->-r requirements.txt (line 4)) (2025.1)
Requirement already satisfied: dimod==0.12.18 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.12.18)
Requirement already satisfied: dwave-cloud-client==0.13.3 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.13.3)
Requirement already satisfied: dwave-gate==0.3.3 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.3.3)
Requirement already satisfied: dwave-hybrid==0.6.13 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.6.13)
Requirement already satisfied: dwave-inspector==0.5.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.5.2)
Requirement already satisfied: dwave-networkx==0.8.16 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.8.16)
Requirement already satisfied: dwave-optimization==0.5.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.5.1)
Requirement already satisfied: dwave-preprocessing==0.6.7 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.6.7)
Requirement already satisfied: dwave-samplers==1.4.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (1.4.0)
Requirement already satisfied: dwave-system==1.29.0 in
```

```
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (1.29.0)
Requirement already satisfied: dwavebinarycsp==0.3.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.3.1)
Requirement already satisfied: minorminer==0.2.17 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.2.17)
Requirement already satisfied: penaltymodel==1.2.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (1.2.0)
Requirement already satisfied: scipy>=1.9 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
qutip==5.1.1->-r requirements.txt (line 7)) (1.15.1)
Requirement already satisfied: dwave-neal>=0.5.7 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pyqubo==1.5.0->-r requirements.txt (line 8)) (0.6.0)
Requirement already satisfied: Deprecated>=1.2.12 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pyqubo==1.5.0->-r requirements.txt (line 8)) (1.2.18)
Requirement already satisfied: six>=1.15.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pyqubo==1.5.0->-r requirements.txt (line 8)) (1.17.0)
Requirement already satisfied: requests<3,>=2.25 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
requests[socks]<3,>=2.25->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (2.32.3)
Requirement already satisfied: urllib3<3,>=1.26 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (2.3.0)
Requirement already satisfied: pydantic<3,>=2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (2.10.6)
Requirement already satisfied: homebase<2,>=1.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (1.0.1)
Requirement already satisfied: click<9,>=7.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (8.1.8)
Requirement already satisfied: plucky<0.5,>=0.4.3 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (0.4.3)
Requirement already satisfied: diskcache<6,>=5.2.1 in
```

```
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (5.6.3)
Requirement already satisfied: werkzeug<4,>=2.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (3.1.3)
Requirement already satisfied: typing-extensions<5,>=4.5.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (4.12.2)
Requirement already satisfied: authlib<2,>=1.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (1.4.1)
Requirement already satisfied: importlib_metadata>=5.0.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (8.6.1)
Requirement already satisfied: orjson>=3.10 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r requirements.txt (line
6)) (3.10.15)
Requirement already satisfied: Flask<4,>=2.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
dwave-inspector==0.5.2->dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6))
(3.1.0)
Requirement already satisfied: fasteners>=0.15 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
minorminer==0.2.17->dwave-ocean-sdk==8.2.0->-r requirements.txt (line 6)) (0.19)
Requirement already satisfied: wrapt<2,>=1.10 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
Deprecated>=1.2.12->pyqubo==1.5.0->-r requirements.txt (line 8)) (1.17.3)
Requirement already satisfied: cryptography in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
authlib<2,>=1.2->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (44.0.0)
Requirement already satisfied: Jinja2>=3.1.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
Flask<4,>=2.2->dwave-inspector==0.5.2->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (3.1.5)
Requirement already satisfied: itsdangerous>=2.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
Flask<4,>=2.2->dwave-inspector==0.5.2->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (2.2.0)
Requirement already satisfied: blinker>=1.9 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
Flask<4,>=2.2->dwave-inspector==0.5.2->dwave-ocean-sdk==8.2.0->-r
```

```
requirements.txt (line 6)) (1.9.0)
Requirement already satisfied: zipp>=3.20 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
importlib_metadata>=5.0.0->dwave-cloud-client==0.13.3->dwave-ocean-
sdk==8.2.0->-r requirements.txt (line 6)) (3.21.0)
Requirement already satisfied: annotated-types>=0.6.0 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pydantic<3,>=2->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (0.7.0)
Requirement already satisfied: pydantic-core==2.27.2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
pydantic<3,>=2->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (2.27.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
requests<3,>=2.25->requests[socks]<3,>=2.25->dwave-cloud-client==0.13.3->dwave-
ocean-sdk==8.2.0->-r requirements.txt (line 6)) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
requests<3,>=2.25->requests[socks]<3,>=2.25->dwave-cloud-client==0.13.3->dwave-
ocean-sdk==8.2.0->-r requirements.txt (line 6)) (3.10)
Requirement already satisfied: certifi>=2017.4.17 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
requests<3,>=2.25->requests[socks]<3,>=2.25->dwave-cloud-client==0.13.3->dwave-
ocean-sdk==8.2.0->-r requirements.txt (line 6)) (2024.12.14)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
requests[socks]<3,>=2.25->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (1.7.1)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
werkzeug<4,>=2.2->dwave-cloud-client==0.13.3->dwave-ocean-sdk==8.2.0->-r
requirements.txt (line 6)) (3.0.2)
Requirement already satisfied: cffi>=1.12 in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
cryptography->authlib<2,>=1.2->dwave-cloud-client==0.13.3->dwave-ocean-
sdk==8.2.0->-r requirements.txt (line 6)) (1.17.1)
Requirement already satisfied: pycparser in
/home/tsmierzchalski/anaconda3/envs/szkolenia/lib/python3.12/site-packages (from
cffi>=1.12->cryptography->authlib<2,>=1.2->dwave-cloud-client==0.13.3->dwave-
ocean-sdk==8.2.0->-r requirements.txt (line 6)) (2.22)
Note: you may need to restart the kernel to use updated packages.
```

3. Poniższą komendę wykonaj w terminalu. Konfiguruje software niezbędny do pracy z komputerami dwave.

dwave setup

Kilka słów o używanych pakietach

- `NumPy` - obliczenia numeryczne, u nas dla efektywnego wykonywania operacji na macierzach
- `Pandas` - obróbka i praca ze zbiorami danych
- `Matplotlib` - wykresy, ogólniej wizualizacje matematyczne
- `Networkx` - działanie na grafach
- `tqdm` - renderowanie paska postępu, dla wygody
- `dwave-ocean-sdk` - komunikacja z dwave i wiele innych przydatnych funkcjonalnalności przydatnych dla naszych modeli, obszerna paczka pakietów

2.3 Dodatkowe wymagania odnośnie GPU

Część szkolenia będzie skupiona na wykorzystaniu możliwości GPU do wspierania obliczeń. Wymagania sprzętowe:

1. Procesor graficzny NVIDIA CUDA o możliwości obliczeniowej (Compute Capability) 3.0 lub wyższej. Niemal wszystkie współczesne karty graficzne spełniają to wymaganie. Listę wspieranych urządzeń znajdziesz tutaj.
2. CUDA Toolkit: v11.2 / v11.3 / v11.4 / v11.5 / v11.6 / v11.7 / v11.8 / v12.0 / v12.1 / v12.2 / v12.3 / v12.4 / v12.5 / v12.6

By sprawdzić wersję CUDA Toolkit wpisz `nvcc --version` w terminalu.

Do interakcji z GPU będzie potrzebny dodatkowy pakiet Python - `CuPy`. Komenda do instalacji `CuPy` różni się w zależności od posiadanej wersji CUDA Toolkit. W przypadku problemów z instalacją pomoc znajdziesz tutaj.

[ ]:
```
# Instalacja CuPy dla v11.2 ~ 11.8

%pip install cupy-cuda11x==13.6
```

[ ]:
```
# Instalacja CuPy dla v12.x

%pip install cupy-cuda12x==13.6
```

3 Dodatkowe środowisko wymagane by wykonać benchmarki

Wykonanie niektórych benchmarków będzie wymagało innej wersji pythona. W konsoli wpisz po kolei:

conda deactivate

zmiana środowiska w konsoli

conda create –name benchmark conda activate benchmark conda install python=3.9

[ ]:
```
%pip install ipykernel
%pip install omnisolver-bruteforce
%pip install git+https://github.com/bqth29/simulated-bifurcation-algorithm.git
```

Kilka słów o używanych pakietach

- `omnisolver-bruteforce`- wysoko zoptymalizowana implementacja algorytmu wyczerpującego przeszukiwania, która używa podobnego algorytmu co nasza implementacja. Tutaj można znaleźć dodatkowe informacje.
- `Simulated Bifurcation for Python` - implementacja algorytmu symulowanej bifurkacji. Tutaj można znaleźć dodatkowe informacje.

Klasyczny model Isinga

Model, wprowadzony na początku lat 20. XX wieku przez Ernsta Isinga, pierwotnie miał być modelem ferromagnetyzmu. Od tego czasu stał się fundamentalnym narzędziędziem do badań przejść fazowych i emergentnych właściwości w różnych układach fizycznych. Co ciekawe, znalazł zastosowania również poza fizyką - znajduje zastosowania w takich dziedzinach jak finanse, dynamika społeczna oraz optymalizacja w zagadnieniach Informatycznych. W tym kursie skupimy się na jednym problemie z nim związanym: znalezieniu stanu podstawowego (tj. stanu o najniższej energii) modelu Isinga oraz widma niskich energii.

Definicja

Rozważmy wektor zmiennych $\bm{s} = (s_1, s_2, \ldots, s_N)$, gdzie $s_i = \pm 1$. Zmienne te nazywane są **spinami** i mogą przyjmować jeden z dwóch stanów: w górę ($+1$) lub w dół (-1). Spiny są rozmieszczone na pewnym grafie prostym G, który opisuje interakcje między nimi. Krawędzie grafu G (oznaczane jako $E(G)$) określają, które spiny oddziałują ze sobą. Siła interakcji (lub *coupling*) między s_i a s_j jest oznaczana jako $J_{ij} \in \mathbb{R}$. Dodatkowo każdy spin poddawany jest działaniu zewnętrznego pola magnetycznego (lub *bias* albo *obciążenie*) $h_i \in \mathbb{R}$, $\bm{h} = (h_1, h_2, \ldots, h_N)$. Energia układu opisana jest następującą funkcją (**Hamiltonianem**):

$$ H(\bm{s}) = \sum_{(i, j) \in E(G)} J_{ij} s_i s_j + \sum_{i=1}^{N} h_i s_i $$

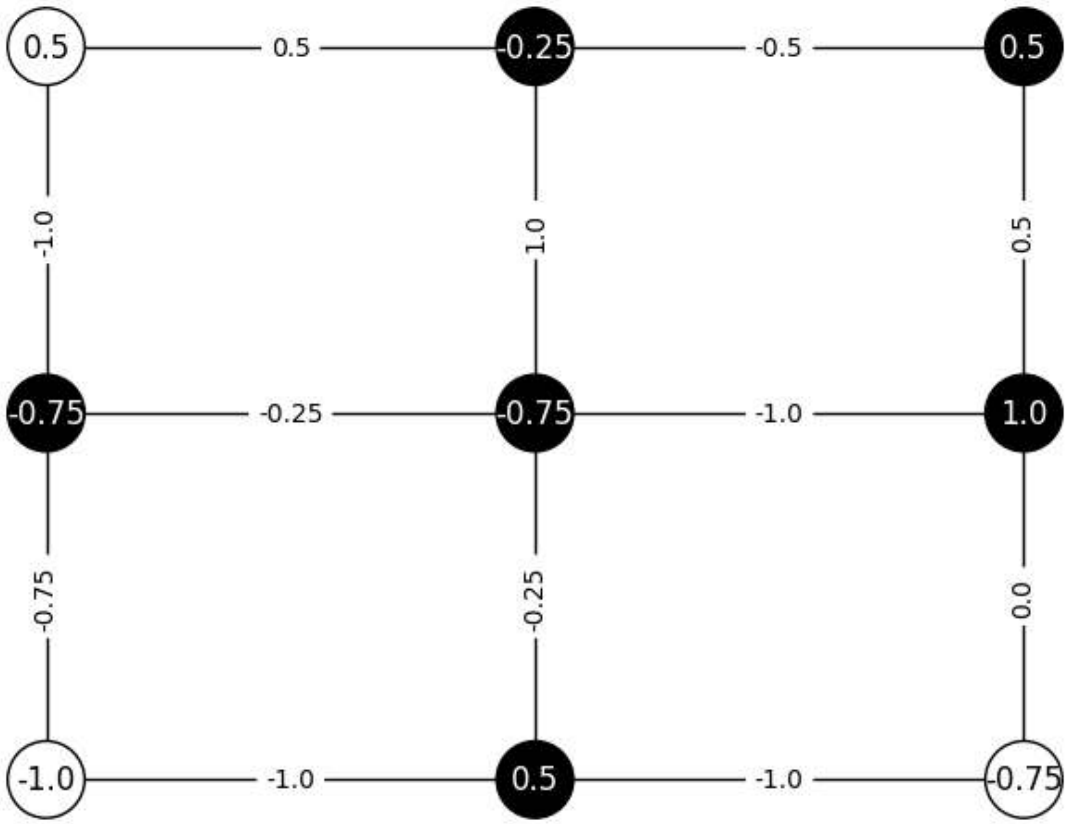
Każda realizacja $\bm{s}$ jest nazywana **stanem** albo **konfiguracją**. Trójkę (G, J, h) nazywamy **instancją**.

Pierwsza instancja

Zaczniemy od utworzenia instancji. Następnie policzymy jej energię ze wzoru i metodą macierzową.

In [6]:
```
# Instancja razem z obrazkiem

import random
import networkx as nx
import matplotlib.pyplot as plt

random.seed(7)

# Tworzymy graf
instance = nx.grid_2d_graph(3, 3)
```

```
# Nadajemy losowe wartości spinu, h i J
nx.set_node_attributes(instance, {node: random.choice([-1, 1]) for node in i
nx.set_node_attributes(instance, {node: random.choice([i / 4 for i in range(
nx.set_edge_attributes(instance, {edge: random.choice([i / 4 for i in range(

# Ustalamy kolor wierzchołków w zalezności od wartości spinu, -1 (dół) czarn
node_colors = ["white" if instance.nodes[node]["spin"] == 1 else "black" for


# Rysujemy instancję, której będziemy używać
pos={node: node for node in instance.nodes}
nx.draw(instance, pos, node_color=node_colors,  node_size=900, font_weight='

# rysujemy wierzchołki i pokazujemy wartość h na nich
node_labels = {node: f"{instance.nodes[node]['h']}" for node in instance.noc
for node in instance.nodes:
    label = f"{instance.nodes[node]['h']}"
    x, y = pos[node]
    font_color = "white" if instance.nodes[node]["spin"] == -1 else "black"
    plt.text(x, y, label, fontsize=12, ha='center', va='center', color=font_
ax = plt.gca() # to get the current axis
ax.collections[0].set_edgecolor("black")

# wpisujemy wartości "J" na krawędziach
edge_labels = {(u, v): f"{data['J']}" for u, v, data in instance.edges(data=
nx.draw_networkx_edge_labels(instance, pos, edge_labels=edge_labels)

plt.show()
```



Energia obliczana ze wzoru $H(\bm{s}) = \sum_{(i, j) \in E(G)} J_{ij} s_i s_j + \sum_{i=1}^{N} h_i s_i$

In [7]:

```
# Obliczenie energii ze wzoru

spins = {node: instance.nodes[node]["spin"] for node in instance.nodes}
h = {node: instance.nodes[node]["h"] for node in instance.nodes}
J = {(u, v): data['J'] for u, v, data in instance.edges(data=True)}









def navie_energy_calculation(spins, J, h, instance):
    # Liczymy w dwóch pętlach
    energy_naive = 0

    # Część liniowa
    for node in instance.nodes:
```

```
        energy_naive += spins[node] * h[node]

    # Część kwadratowa
    for (u, v) in instance.edges:
        energy_naive += spins[u] * spins[v] * J[(u, v)]

    return energy_naive

energy_naive = navie_energy_calculation(spins, J, h, instance)

print(f"Wartość energii wynosi: {energy_naive}")
```

```
Wartość energii wynosi: 1.25
```

Metoda macierzowa

Z punktu widzenia efektywności obliczeń, warto skorzystać z narzędzi algebry liniowej. Rozważmy macierz sąsiedztwa grafu G, przyjmując każdy J_{ij} jako wagę przypisaną do krawędzi. Ponieważ graf G jest grafem prostym, jest on grafem nieskierowanym, więc wartości $J_{ij} = J_{ji}$. Symetria względem przekątnej pozwala nam na uproszczenia (i uniknięcie zbędnej pracy) - możemy zająć się jedynie elementami J_{ij}, gdzie $i < j$, to znaczy macierzą górnotrójkątną. Zwyczajowo oznaczaną jako $\bm{J}$. Wówczas równanie hamiltonianu $H(\bm{s})$ przyjmuje postać:

$$ H(\bm{s}) = \bm{s}\bm{J}\bm{s}^{T} + \bm{h}\bm{s}^{T} $$

Równoważnie, jeżeli J nie jest macierzą górnotrójkątną, tylko hermitowską, to Hamiltonian wygląda następująco:

$$ H(\bm{s}) = \frac{1}{2}\bm{s}\bm{J}\bm{s}^{T} + \bm{h}\bm{s}^{T} $$

W literaturze można jeszcze spotkać następujące sformułowania hamiltonianu Isinga:

$$ H(\bm{s}) = -\bm{s}\bm{J}\bm{s}^{T} - \bm{h}\bm{s}^{T} $$

$$ H(\bm{s}) = -\frac{1}{2}\bm{s}\bm{J}\bm{s}^{T} - \bm{h}\bm{s}^{T} $$

Wszystkie te postacie hamiltonianu są matematycznie równoważne. Będziemy używać każdej z nich w zależności od tego które sformułowanie jest dla danego problemu wygodniejsze.

W zaprezentowany kodzie domyślnie jest używana ostania z zaprezentowanych konwencji (z "$-\frac{1}{2}$"). W wyżarzaczach kwantowych domyslnie stosowana jest pierwsza zaprazentowana konwencja.

In [8]:
```
# Obiczenie energii w wersji macierzowej
import numpy as np

# Zamiana grafowej reprezentacji na macierzową.
```

```python
# UWAGA: od wersji python 3.7 słowniki zachowują kolejność, więc możemy bezp
def dict_to_vect(d: dict):
    return np.array(list(d.values()))

spins_vector = dict_to_vect(spins)
h_vector = dict_to_vect(h)

# Macierz górnotrójkątna J
def dict_to_matrix(instance: nx.Graph, J: dict):
    n = len(instance)
    J_matrix = np.zeros(shape=(n,n))
    renum = {node: idx for idx, node in enumerate(instance.nodes)}
    for (u, v), J_value in J.items():
        i = renum[u]
        j = renum[v]
        if i > j:
            i, j = j, i
        J_matrix[i, j] = J_value
    return J_matrix

J_matrix = dict_to_matrix(instance, J)









# Liczymy energię z kolejnego wzoru

energy_vectorized = spins_vector @ J_matrix @ spins_vector.T + spins_vector


print(f"Wartość energii wynosi: {energy_vectorized}")
```

```
Wartość energii wynosi: 1.25
```

Czym się różnią te metody?

Chociaż mamy do czynienia z różnymi przedstawieniami tej samej funkcji, dającymi takie same wyniki, to ich efektywność znacznie się różni. O ile wykonanie obliczeń w

obu przypadkach było bardzo szybkie, to dla dużych instancji liczenie energii metodą macierzową jest zauważalnie szybsze. Różnice wywołane są wykorzystaniem biblioteki NumPy, która posiada efektywne niskopoziomowe implementacje obliczeń na macierzach.

In [11]:
```
import timeit
import networkx as nx

# Przygotujmy dużą losową instancję

big_instance = nx.complete_graph(1000)

# Nadajemy losowe wartości spinu, h i J
nx.set_node_attributes(big_instance, {node: random.choice([-1, 1]) for node
nx.set_node_attributes(big_instance, {node: random.choice([i / 4 for i in ra
nx.set_edge_attributes(big_instance, {edge: random.choice([i / 4 for i in ra

# Powtarzamy poprzednie procedury
spins_big = {node: big_instance.nodes[node]["spin"] for node in big_instance
h_big = {node: big_instance.nodes[node]["h"] for node in big_instance.nodes}
J_big = {(u, v): data['J'] for u, v, data in big_instance.edges(data=True)}

spins_big_vector = dict_to_vect(spins_big)
h_big_vector = dict_to_vect(h_big)
J_big_matrix = dict_to_matrix(big_instance, J_big)

num_runs = 10
time1 = timeit.timeit(lambda: navie_energy_calculation(spins_big,J_big, h_bi
time2 = timeit.timeit(lambda: spins_big_vector @ J_big_matrix @ spins_big_ve

print(f"Przeciętny czas obliczeń dla metody naiwnej używając {num_runs} wywo
print(f"Przeciętny czas obliczeń dla metody macierzowej używając {num_runs}
print(f"W analizowanym przypadku metoda macierzowa jest {time1/time2:.0f} ra
```

```
Przeciętny czas obliczeń dla metody naiwnej używając 10 wywołań: 0.013250 se
kund
Przeciętny czas obliczeń dla metody macierzowej używając 10 wywołań: 0.00001
7 sekund
W analizowanym przypadku metoda macierzowa jest 769 razy szybsza
```

W dalszej części tego szkolenia będziemy używać metody macierzowej do liczenia energii!

Gradient

W omiawianych algorytmach pojawia się gradient Hamiltonianu Isinga. Dla macierzy hermitowskiej J oraz przy konwencji:

$$ H(\bm{s}) = -\frac{1}{2}\bm{s}\bm{J}\bm{s}^{T} - \bm{h}\bm{s}^{T} $$

Gradient wygląda następująco:

$$ \nabla H(\bm{s}) = -\bm{J}\bm{s} -h $$

Wyprowadzenie

Pochodne funkcji macierzowych i wektorowych zachowują się podobnie do pochodnych funkcji skalarnych. Zainteresowanym polecam podręcznik The Matrix Cookbook.

Warto wspomnieć, że w ogólności: $$ f = \bm{x} \bm{A} \bm{x}^T + \bm{b}\bm{x}^T $$ $$ \nabla f = \frac{\partial f}{\partial \bm{x}} = (\bm{A} + \bm{A}^T)\bm{x} + \bm{b} $$

Interpretacja fizyczna modelu Isinga

Tradycyjnie w fizyce model Isinga opisuje się na d-wymiarowej sieci (w naszej definicji graf G). W modelu ferromagnetyzmu, będącym pierwotnym badaniem Isinga, można wyobrazić sobie węzły sieci jako atomy materiału magnetycznego. Każdy atom posiada moment magnetyczny (spin), który może być skierowany albo „w górę", albo „w dół". W ten sposób, jeśli wszystkie węzły są skierowane „w górę" lub „w dół", mamy do czynienia z ferromagnetykiem. Z kolei, jeśli wszystkie sąsiednie węzły są skierowane w przeciwnych kierunkach, powstaje antyferromagnetyk, charakteryzujący się brakiem (lub zanikiem) magnetyzacji. W tej interpretacji h_i reprezentuje zewnętrzne pole magnetyczne, które wymusza skierowanie momentu magnetycznego w określonym kierunku. Oddziaływania między spinami J_i mogą być ferromagnetyczne, jeśli sprzyjają skierowaniu sąsiadujących węzłów w tym samym kierunku; w przeciwnym razie nazywane są antyferromagnetycznymi. Stan, w którym spiny są losowo skierowane w górę lub w dół, często nazywany jest szkłem spinowym (spin glass).

Stany niskoenergetyczne

Jednym z podstawowych problemów dotyczących modelu Isinga jest poszukiwanie **stanu podstawowego** - stanu o najmniejszej energii. Zatem interesuje nas minimalizacja wartości Hamiltonianu:

$$ \min_{\bm{s}} H(\bm{s}) $$

Problem ten jest istotny, ponieważ stan podstawowy określa najbardziej stabilne ułożenie całego układu i pozwala zrozumieć jego własności fizycznee, takie jak przejścia fazowe czy zachowanie w pobliżu temperatury krytycznej. Co więcej, znalezienie konfiguracji minimalnej energii ma znaczenie wykraczające poza fizykę – wiele trudnych problemów optymalizacyjnych w informatyce, biologii czy naukach społecznych można sformułować w języku modelu Isinga, dlatego metody poszukiwania stanu podstawowego znajdują szerokie zastosowania praktyczne.

W ogólności jest to problem NP-trudny. W skrócie, to oznacza że nie istnieją żadne znane algorytmy które w rozsądnym czasie potrafią znaleść stan podstawowy. Ponadto, każdy problem który jest w klasie NP można efektywnie przedstawić w języku modelu Isinga.

Przykład - Problem podziału jako model Isinga

Formalnie: mając dany zbiór N dodatnich liczb całkowitych $U = \lbrace n_1, \ldots, n_N \rbrace$, chcemy znaleźć dwa rozłączne podzbiory $X, Y \subset U$, $X \cap Y = \emptyset$, $X \cup Y = U$ takie, że sumy ich elementów są sobie równe tj. $$ \sum_{n \in X}n = \sum_{n' \in Y} n' \Leftrightarrow \sum_{n \in X}n - \sum_{n' \in Y} n'=0 $$ bądź też różnica jest minimalna (nie zawsze rozwiązanie optymalne istnieje). Hamiltonian przyjmuje postać

$$ H = A (\sum_{i=1}^N n_i s_i)^2 $$

gdzie $n_i \in U$, $s_i = \pm 1$ oraz $A > 0$ jest pewną stałą. Widać, że optymalne rozwiązanie $H=0$ można osiągnąć tylko wtedy, gdy suma n_i stojących przy spinach $+1$ jest równa sumie n_i stojących przy spinach -1.

Podany Hamiltonian H jest Hamiltonianem Isinga, widać to po pewnych przekształceniach

$$ H = A (\sum_{i=1}^N n_i s_i)^2 = A(\sum_{i,j} 2 n_i n_j s_i s_j + \sum_{i=1}^N n_i^2 s_i^2) = A \sum_{i,j} J_{ij} s_i s_j + C $$

gdzie $C=A\sum_{i=1}^N n_i^2 s_i^2$ jest pewną stałą, która nie ma wpływu na rozwiązanie, a $J_{ij} = 2 n_i n_j$.

O problemie podziału mniej formalnie i wyprowadzenie Hamiltonianu

Problem podziału najłatwiej będzie wyjaśnić na życiowym przykładzie - podziale spadku. Na majątek składa się biużuteria, książki, antyki. Mamy dwóch spadkobierców, każdy ma otrzymać taką samą wartość majątku. Zakładamy, że przedmiotów nie można "rozmieniać" ani sprzedać, należy rozdać cały majątek. Jak dokonać tego najbardziej sprawiedliwie? Dobrać podział tak, aby różnica była możliwie najmniejsza.

Spróbujemy przedstawić ten problem za pomocą Hamiltonianu. Optymalne rozwiązanie zakłada równy podział (różnica wynosi 0), natomiast nas będzie interesować minimalna różnica: $$ \vert\sum_{n \in X}n - \sum_{n' \in Y} n'\vert= \min \Leftrightarrow \vert\sum_{n \in X}1\cdot n + \sum_{n' \in Y} (-1)\cdot n'\vert=\min $$ $$ \vert\sum_{n \in S}(\pm1)n\vert= \vert\sum_{i=1}^N n_i s_i\vert=\min \Leftrightarrow (\sum_{i=1}^N n_i s_i)^2=\min, $$ gdzie $s_i=1$ oznacza, że $n_i \in X$, a $s_i=-1$, że $n_i\in Y$.

In [10]:
```python
## Przykładowy problem podziału

from dimod import BinaryQuadraticModel, ExactSolver
from pyqubo import Array

solver = ExactSolver()

# zadany zbiór
U = [2, 3, 4, 5, 6]

# tworzymy model oraz hamiltonian
spins = Array.create("spins", shape=len(U), vartype="SPIN")

H = sum(n * s for n, s in  zip(U, spins))**2
model = H.compile()

linear, quadratic, offset = model.to_ising()

# rozwiązujemy problem
bqm = BinaryQuadraticModel(linear, quadratic, offset, vartype="SPIN")
sampleset = solver.sample_ising(linear, quadratic)
print(sampleset.lowest(0))

# sprawdzamy rozwiązanie, pamiętając o stałej
energy = sampleset.first.energy

if energy + offset == 0:
    print("Prawidłowe rozwiązanie!")
else:
    print("błąd")
```

```
  spins[0] spins[1] spins[2] spins[3] spins[4] energy num_oc.
0       -1       -1       +1       -1       +1  -90.0       1
1       +1       +1       -1       +1       -1  -90.0       1
['SPIN', 2 rows, 2 samples, 5 variables]
Prawidłowe rozwiązanie!
```

Rozwiązanie problemu Isinga?

Istnieje wiele algorytmów przeznaczonych do rozwiązania tego problemu. Na tym szkoleniu skupimy się na wybranych algorytmach klasycznych takich jak:

- Wyczerpujące przeszukiwanie
- Symulowane wyżarzanie
- Wyżarzanie równoległe
- Branch and bound
- Symulowana bifurkacja

Oraz na metodzie **wyżarzania kwantowego** razem z zastosowaniem wyżarzaczy kwantowych firmy D-Wave.

Literatura

- Dowolny podręcznik poświęcony mechanice statystycznej (ex. Luca Peliti: Statistical Mechanics in a Nutshell, R.J. Baxter: Exactly Solved Models in Statistical Mechanics, H.E. Stanley: Introduction to Phase Transitions and Critical Phenomena etc.)
- Cipra, B. A. (1987). An introduction to the Ising model. *The American Mathematical Monthly*, 94(10), 937-959.
- Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2

Model QUBO

Quadratic Unconstrained Binary Optimization (QUBO) to model do rozwiązywania problemów optymalizacji kombinatorycznej. Jak sugeruje nazwa, dotyczy on modeli optymalizacyjnych (O), w których występują zależności kwadratowe (Q) między zmiennymi binarnymi (B), brak ograniczeń (U). Co zaskakujące, warunki QUBO spełnia duża klasa problemów, takich jak MAX-CUT, problem komiwojażera czy harmonogramowanie zadań.

Model QUBO - definicja

Dla danej symetrycznej macierzy Q, należy znaleźć binarny wektor $\bm{x}^*$, taki że:

$$ \bm{x}^* = \arg \min_x \bm{x}Q\bm{x}^T = \sum_i \sum_j Q_{ij} x_i x_j $$

Często przekształca się macierz Q do formy górnotrójkątnej - dla wszystkich i, j, gdzie $j > i$, zastępuje się element powyżej przekątnej Q_{ij} przez $Q_{ij} + Q_{ji}$. Następnie wszystkie elementy poniżej przekątnej Q_{ji} dla $j < i$ zamienia się na 0.

Warto zauważyć, że dla zmiennych binarnych zachodzi $x^2 = x$ ($0^2=0$, $1^2=1$). Pozwala to podzielić QUBO na część liniową znajdującą się na diagonali oraz część kwadratową:

$$ \bm{x}Q\bm{x}^T = \sum_i Q_{ii} x_i + \sum_{i \neq j} Q_{ij} x_i x_j $$

Rozwiązywanie QUBO - przykład

Podążając za przykładem spróbujemy lepiej zrozumieć, w jaki sposób można przekształcić problemy w formę QUBO. Na początek rozważmy następujący problem optymalizacyjny - chcemy zminimalizować wartość wyrażenia: $$ y = -5x_1 -3x_2 -8x_3 - 6x_4 + 4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4 $$

Obserwacje:

1. y składa się z części liniowej $-5x_1 -3x_2 -8x_3 - 6x_4$ oraz części kwadratowej $4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4$.
2. Ponieważ dla zmiennych binarnych $x_i = x_i^2$, w części liniowej można zastąpić każdą zmienną jej kwadratem:
   $$ -5x_1^2 -3x_2^2 -8x_3^2 - 6x_4^2 $$

co daje postać homogeniczną: $$ y=-5x_1^2 -3x_2^2 -8x_3^2 - 6x_4^2+4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4 $$

3. Następnie możemy przekształcić model do następującej postaci macierzowej: $$ y = [x_1 \, x_2 \, x_3 \, x_4] \begin{bmatrix} -5 & 4 & 8 & 0 \\ 0 & -3 & 2 & 0 \\ 0 & 0 & -8 & 10 \\ 0 & 0 & 0 & -6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} $$

lub krócej:
$$ y = \arg \min_x \bm{x}Q\bm{x}^T $$
Należy zauważyć, że stosujemy tu konwencję zachowania Q jako macierzy górnotrójkątnej.

4. Poza wymogiem binarności zmiennych decyzyjnych, QUBO jest modelem nieograniczonym, gdzie wszystkie dane problemu są zawarte w macierzy Q. Fakt ten sprawia, że model QUBO jest atrakcyjną formą dla wielu problemów optymalizacyjnych.

In [1]:
```python
# Metota naiwnego wyczerpującego przeszukiwania


from dimod import ExactSolver
import numpy as np

Q = np.array([[-5, 4, 8, 0], [0, -3, 2, 0], [0, 0, -8, 10], [0, 0, 0, -6]])

sampleset = ExactSolver().sample_qubo(Q)
solution_vector = list(sampleset.first.sample.values())
solution = sampleset.first.energy

print("rozwiązanie:", end = " ")
for idx, i in enumerate(solution_vector):
    print(f"x_{idx + 1} = {i},", end = " ")
print(f"\ny_min = {solution}")
```

```
rozwiązanie: x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1,
y_min = -11.0
```

Podobieństwo między QUBO a modelem Isinga

Nie sposób nie zauważyć, że wyrażenia opisujące QUBO i model Isinga są bliźniaczo podobne

$$ H(\textbf{s}) = \sum_{(i,j)} J_{ij} s_i s_j + \sum_i h_i s_i \quad \text{oraz} \quad \bm{x}Q\bm{x}^T = \sum_{i \neq j} Q_{ij} x_i x_j + \sum_i Q_{ii} x_i $$ gdzie zakładamy, że $\textbf{J}$ i $\textbf{Q}$ są macierzami górnotrójkątnymi, $i = 1, \ldots, N$, a różnicą są wartości $s_i = \pm 1$, $x_i\in\lbrace 0, 1\rbrace$.

W rzeczywistości rozwiązanie QUBO i znajdowanie stanów podstawowych modelu Isinga to równoważne problemy! Można je stosunkowo łatwo przekształcić między sobą za pomocą następującego podstawienia:

$$ x_i = \frac{s_i + 1}{2} $$
$$ s_i = 2x_i - 1 $$

Podstawiając te zmienne do odpowiednich wzorów, otrzymujemy:

$$ Q_{ij} = 4 J_{ij} $$
$$ Q_{ii} = 2h_i - 2\sum_{\mathcal{N}(i)} J_{ij} $$

$$ \text{offset} = \sum_{(i,j)}J_{ij} - \sum_{i}h_i $$

oraz

$$ J_{ij} = \frac{1}{4} Q_{ij} $$
$$ h_{i} = \frac{1}{2}Q_{ii} + \frac{1}{4} \sum_{j \neq i} Q_{ij} $$

$$ \text{offset} = \frac{1}{2}\sum_{i}Q_{ii} + \frac{1}{4}\sum_{i < j}Q_{ij} $$

Należy pamiętać, że transformacja generuje pewną dodatkową stałą, więc uzyskane wartości funkcji celu nie mogą być porównywane bezpośrednio.

Zamiana pomiędzy modelami

In [2]:
```python
import numpy as np
rng = np.random.default_rng(seed=42)

# Przykładowy model Isinga
N = 15
h = {i: 0.5 for i in range(N)}
J = {(i, i+1): -1 for i in range(N-1)}

# Ising to qubo

Q = {}



# część liniowa
for i in h.keys():
    if i != 0 and i != N-1:
        Q[(i,i)] = 2*h[i] - 2*(J[(i,i+1)] + J[(i-1, i)])
    elif i == 0:
         Q[(i,i)] = 2*h[i] - 2*(J[(i,i+1)])
    else:
        Q[(i,i)] = 2*h[i] - 2*(J[(i-1,i)])
```

```
# część kwadratowa
for (i, j) in J.keys():
    Q[(i,j)] = 4 * J[(i,j)]

offset = sum(list(J.values())) - sum(list(h.values()))


state_ising = [rng.choice([-1, 1]) for i in range(N)]
state_qubo = [(s+1)/2 for s in state_ising]

energy_ising = sum([coupling * state_ising[i] * state_ising[j] for (i, j), c
                sum(bias * state_ising[i] for i, bias in h.items())
energy_qubo = sum([x * state_qubo[i] * state_qubo[j] for (i, j), x in Q.item

print("offset: ", offset)
print("energia Isinga: ", energy_ising)
print("energia QUBO: ", energy_qubo)
print("energia QUBO + offset", energy_qubo + offset)
```

```
offset:  -21.5
energia Isinga:  1.5
energia QUBO:  23.0
energia QUBO + offset 1.5
```

Na szczęście istnieje wiele narzędzi pozwalających automatycznie przechodzic pomiędzy modelami. Poniżej zostanie zaprezentowane jak to zrobić za pomocą paczki `dimod` i zawartej w niej klasy `BinaryQadraticModel`

In [3]:
```
# Zamiana między modelami.
import numpy as np
from dimod import BinaryQuadraticModel

rng = np.random.default_rng(seed=42)

# Przykładowy model Isinga
N = 15
h = {i: 0.5 for i in range(N)}
J = {(i, i+1): -1 for i in range(N-1)}

# Utworzenie BQM
bqm = BinaryQuadraticModel(h, J, vartype="SPIN")


# zamiana modelu Isinga na QUBO
# Warto pamiętać, że zamiana daje nam zarówno model w nowej formie jak i tzw

QUBO, offset = bqm.to_qubo()

state_ising = [rng.choice([-1, 1]) for i in range(N)]
state_qubo = [(s+1)/2 for s in state_ising]


energy_ising = sum([coupling * state_ising[i] * state_ising[j] for (i, j), c
                sum(bias * state_ising[i] for i, bias in h.items())
energy_qubo = sum([x * state_qubo[i] * state_qubo[j] for (i, j), x in QUBO.i
```

```
print("offset: ", offset)
print("energia Isinga: ", energy_ising)
print("energia QUBO: ", energy_qubo)
print("energia QUBO + offset", energy_qubo + offset)

# Tak samo można zamienić QUBO na Isinga
# ćwiczenie dla grupy
```

```
offset:  -21.5
energia Isinga:  1.5
energia QUBO:  23.0
energia QUBO + offset 1.5
```

Przykład - dyskretny problem plecakowy

Dyskretny problem plecakowy (ang. *0-1 Knapsack Problem*) należy do klasycznych problemów optymalizacji kombinatorycznej. Dane są:

- zbiór przedmiotów $i = 1,2,\dots,n$,
- wartość każdego przedmiotu v_i,
- waga każdego przedmiotu w_i,
- pojemność plecaka W.

Celem jest maksymalizacja łącznej wartości wybranych przedmiotów przy zachowaniu ograniczenia pojemności plecaka:

Formalna definicja

Dla każdego przedmiotu podejmujemy decyzję binarną:

$$ x_i \in \{0,1\}, $$

gdzie:

- $x_i = 1$ oznacza, że przedmiot został wybrany,
- $x_i = 0$ oznacza, że przedmiot nie został wybrany.

$$ \max \sum_{i=1}^{n} v_i x_i $$

przy warunku:

$$ \sum_{i=1}^{n} w_i x_i \leq W. $$

Jest to więc problem wyboru takiego podzbioru przedmiotów, który daje jak największą wartość, ale nie przekracza dopuszczalnej wagi.

Przejście do formulacji QUBO

Model QUBO (*Quadratic Unconstrained Binary Optimization*) wymaga zapisania problemu w postaci funkcji kwadratowej zmiennych binarnych bez jawnych ograniczeń. Oznacza to, że ograniczenie pojemności trzeba włączyć do funkcji celu w postaci składnika karnego.

Standardowa postać QUBO ma postać:

$$ \min \; x^T Q x, \qquad x \in \{0,1\}^m. $$

Ponieważ problem plecakowy jest sformułowany jako zadanie maksymalizacyjne z ograniczeniem, przepisujemy go do postaci minimalizacyjnej i dodajemy karę za naruszenie ograniczenia.

Wprowadzenie zmiennych pomocniczych

Aby zamienić nierówność

$$ \sum_{i=1}^{n} w_i x_i \leq W $$

na równość, wprowadzamy zmienną luzu $ s $, taką że:

$$ \sum_{i=1}^{n} w_i x_i + s = W, \qquad s \geq 0. $$

Ponieważ w modelu QUBO wszystkie zmienne muszą być binarne, zmienną $ s $ kodujemy binarnie:

$$ s = \sum_{k=0}^{K} 2^k y_k, \qquad y_k \in \{0,1\}. $$

Liczba bitów $K+1$ powinna być wystarczająca do reprezentacji wartości od 0 do W.

Po podstawieniu otrzymujemy warunek:

$$ \sum_{i=1}^{n} w_i x_i + \sum_{k=0}^{K} 2^k y_k = W. $$

Przekształcając:

$$ \sum_{i=1}^{n} w_i x_i + \sum_{k=0}^{K} 2^k y_k - W = 0. $$

Funkcja celu z karą

Chcemy maksymalizować wartość przedmiotów, więc w wersji minimalizacyjnej używamy składnika:

$$ - \sum_{i=1}^{n} v_i x_i. $$

Aby wymusić spełnienie ograniczenia, dodajemy karę kwadratową:

$$ A \left(\sum_{i=1}^{n} w_i x_i + \sum_{k=0}^{K} 2^k y_k - W \right)^2, $$

gdzie $A > 0$ jest współczynnikiem kary (często też używa się oznaczenia P).

Ostatecznie otrzymujemy funkcję QUBO:

$$ \min \; A \left(\sum_{i=1}^{n} w_i x_i + \sum_{k=0}^{K} 2^k y_k - W \right)^2 - B \sum_{i=1}^{n} v_i x_i, $$

gdzie:

- A odpowiada za egzekwowanie ograniczenia pojemności,
- B wzmacnia wpływ maksymalizacji wartości.

Zazwyczaj wybiera się A istotnie większe od B $(A \gg B)$, aby rozwiązania niezgodne z ograniczeniem były silnie penalizowane.

In [ ]:
```python
items = [
    {"name": "A", "weight": 4, "value": 8},
    {"name": "B", "weight": 5, "value": 10},
    {"name": "C", "weight": 2, "value": 5},
    {"name": "D", "weight": 3, "value": 6},
]

W = 9   # pojemność plecaka

weights = [item["weight"] for item in items]
values = [item["value"] for item in items]
n = len(items)

# Współczynniki modelu
A = 20  # kara za niespełnienie ograniczenia
B = 1
```

Bibliografia

- Lucas, A. (2014). Ising formulations of many NP problems. *Frontiers in Physics*, Volume 2
- Glover, F., Kochenberger, G., Hennig, R. et al. (2022). Quantum bridge analytics I: a tutorial on formulating and using QUBO models. *Annals of Operations Research* **314**, 141–183

03_Przykłady_kodowania_problemow_QUBO_Ising

May 26, 2026

1 Kodowanie dyskretnych problemów optymalizacyjnych za pomocą QUBO / Isinga.

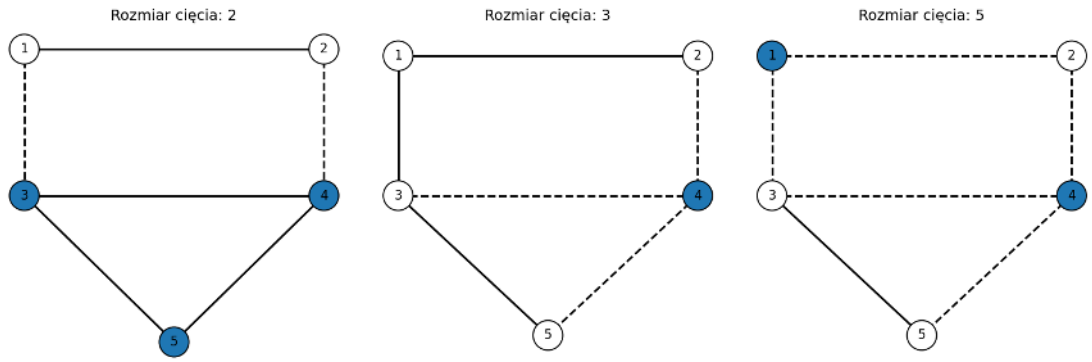
1.1 Przykład - zapisanie problemu MAX-CUT jako QUBO (Zadanie Domowe)

Problem Max Cut jest jednym z najbardziej znanych problemów optymalizacji kombinatorycznej. Dla danego grafu nieskierowanego $G(V, E)$ ze zbiorem wierzchołków V oraz zbiorem krawędzi E, problem Max Cut polega na podziale zbioru V na dwa podzbiory, S, T, $S \cup T = V$, tak aby liczba krawędzi łączących te podzbiory (tzw. przeciętych przez cięcie) była jak największa.

[2]:
```
import networkx as nx
import matplotlib.pyplot as plt

# Utwórz graf
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4, 5])
G.add_edges_from([
    (1, 2),  # krawędź górna
    (1, 3),  # krawędź pionowa lewa
    (2, 4),  # krawędź pionowa prawa
    (3, 4),  # krawędź środkowa pozioma
    (3, 5),  # przekątna lewa
    (4, 5),  # przekątna prawa
])

# Stałe pozycje węzłów dla powtarzalnego układu
pos = {
    1: (0.0,  1.0),
    2: (1.0,  1.0),
    3: (0.0,  0.0),
    4: (1.0,  0.0),
    5: (0.5, -1.0),
}

# Definicja trzech rozcięć (zbiory S) wraz z tytułami
cuts = [
    ({3, 4, 5}, "Rozmiar cięcia: 2"),
    ({4},       "Rozmiar cięcia: 3"),
```

```
    ({1, 4},    "Rozmiar cięcia: 5"),
]

fig, axes = plt.subplots(ncols=3, figsize=(15, 5))
for ax, (S, title) in zip(axes, cuts):
    T = set(G.nodes()) - S
    # Podział krawędzi na: wewnątrz S, wewnątrz T oraz krawędzie cięcia
    edges_inside_S = [(u, v) for u, v in G.edges() if u in S and v in S]
    edges_inside_T = [(u, v) for u, v in G.edges() if u in T and v in T]
    edges_cut      = [(u, v) for u, v in G.edges() if (u in S) ^ (v in S)]

    # Rysuj węzły: niebieskie jeśli w S, białe w przeciwnym wypadku
    node_colors = ["tab:blue" if n in S else "white" for n in G.nodes()]
    nx.draw_networkx_nodes(
        G, pos,
        nodelist=G.nodes(),
        node_color=node_colors,
        edgecolors="black",
        linewidths=1.2,
        node_size=800,
        ax=ax
    )
    nx.draw_networkx_labels(
        G, pos,
        labels={n: str(n) for n in G.nodes()},
        font_color="black",
        font_size=12,
        ax=ax
    )

    # Rysuj krawędzie wewnątrz T
    nx.draw_networkx_edges(
        G, pos,
        edgelist=edges_inside_T,
        style="solid",
        edge_color="black",
        width=2.0,
        ax=ax
    )
    # Rysuj krawędzie wewnątrz S
    nx.draw_networkx_edges(
        G, pos,
        edgelist=edges_inside_S,
        style="solid",
        edge_color="black",
        width=2.0,
        ax=ax
```

```
    )
    # Rysuj krawędzie cięcia
    nx.draw_networkx_edges(
        G, pos,
        edgelist=edges_cut,
        style="dashed",
        edge_color="black",
        width=2.0,
        ax=ax
    )

    ax.set_title(title, fontsize=14)
    ax.set_axis_off()

plt.tight_layout()
plt.show()
```



1.1.1 1. Zmienne binarne

Dla każdego wierzchołka $i \in V$ wprowadzamy zmienną binarną:

$$x_i \in \{0, 1\},$$

gdzie:

- $x_i = 0$ oznacza, że wierzchołek i należy do pierwszego podzbioru,
- $x_i = 1$ oznacza, że wierzchołek i należy do drugiego podzbioru.

Krawędź (i, j) należy do cięcia wtedy i tylko wtedy, gdy końce tej krawędzi znajdują się po przeciwnych stronach, czyli gdy:

$$x_i \neq x_j.$$

1.1.2 2. Funkcja celu

Dla zmiennych binarnych warunek $x_i \neq x_j$ można zapisać za pomocą wyrażenia:

$$x_i + x_j - 2x_i x_j.$$

Sprawdźmy:

- jeśli $x_i = x_j = 0$, to wartość wynosi 0,
- jeśli $x_i = 0, x_j = 1$, to wartość wynosi 1,
- jeśli $x_i = 1, x_j = 0$, to wartość wynosi 1,
- jeśli $x_i = x_j = 1$, to wartość wynosi 0.

1.1.3 3. Przejście do formulacji QUBO

Model QUBO wymaga zapisania zadania w postaci minimalizacji funkcji kwadratowej zmiennych binarnych:

$$\min\ x^T Q x, \qquad x \in \{0,1\}^n.$$

Ponieważ problem Max-Cut jest zadaniem maksymalizacyjnym, przechodzimy do wersji minimalizacyjnej przez zmianę znaku funkcji celu:

$$\min \sum_{(i,j)\in E} \left(-x_i - x_j + 2x_i x_j\right).$$

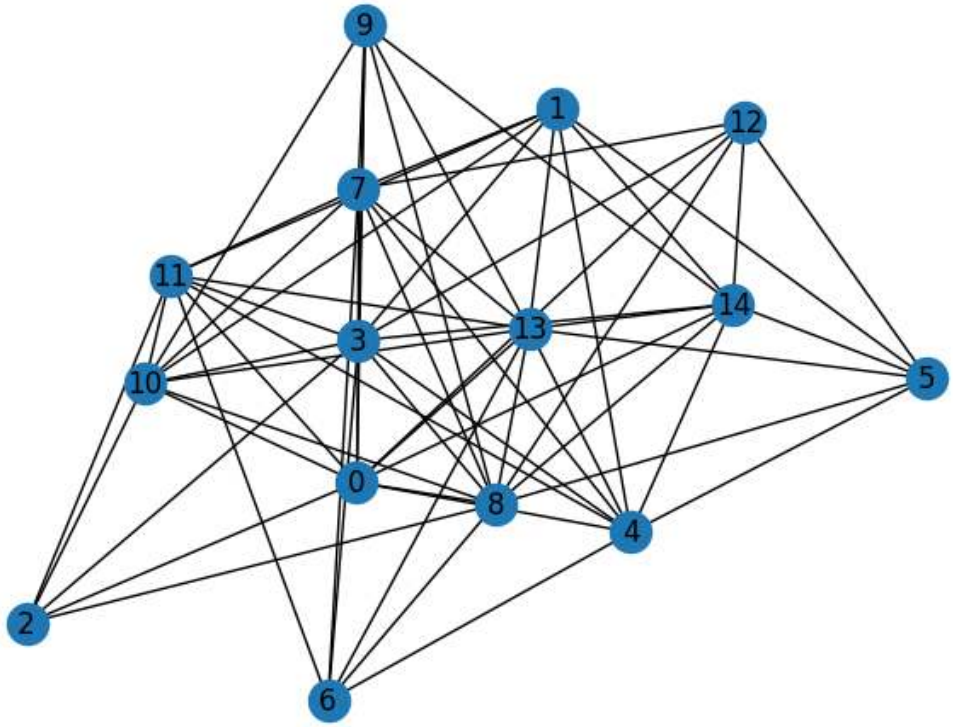
Jest to już funkcja kwadratowa zmiennych binarnych, a więc poprawna postać QUBO.

Interpretacja współczynników W powyższej formulacji:

- składniki liniowe $-x_i$ i $-x_j$ premiują przypisanie wierzchołków do jednej z części,
- składnik kwadratowy $x_i x_j$ koryguje tę premię tak, aby największą wartość uzyskiwały rozwiązania, w których końce krawędzi znajdują się w różnych częściach.

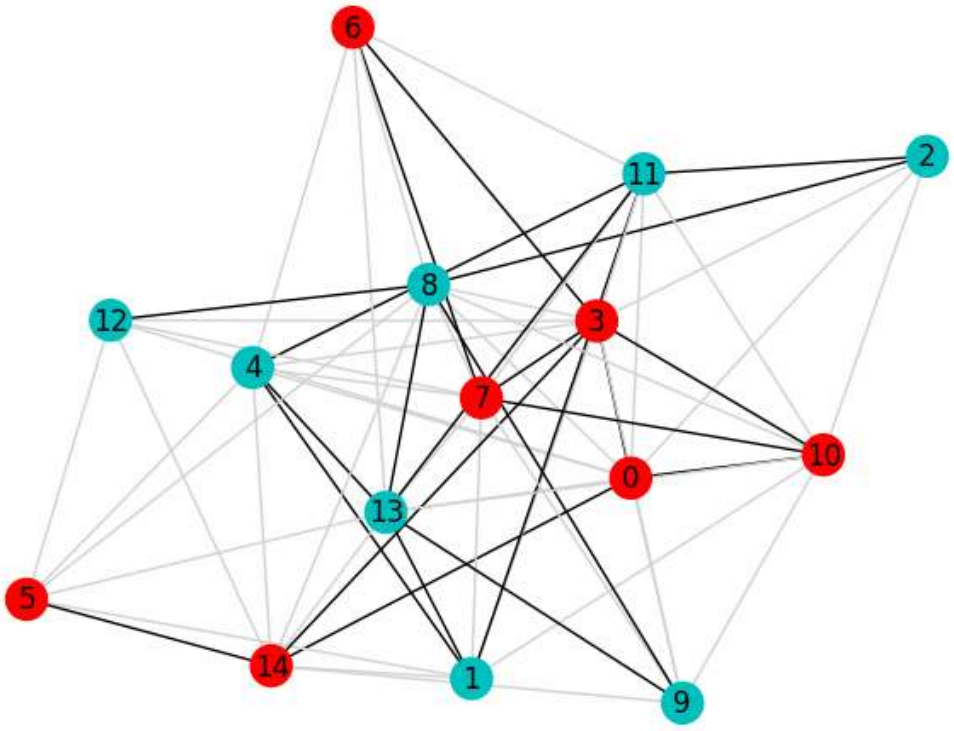
Najważniejsze jest to, że nie są potrzebne żadne dodatkowe ograniczenia ani zmienne pomocnicze. Problem Max-Cut w naturalny sposób prowadzi do funkcji kwadratowej nad zmiennymi binarnymi.

[3]:
```
# Przykład grafu losowego na 15 wierzchołkach
import numpy as np
import networkx as nx

graph = nx.erdos_renyi_graph(n=15, p=0.6, seed=42)
nx.draw(graph, with_labels=True)
```



[5]:
```
# Rozwiązanie problemu
from collections import defaultdict
from math import inf
from itertools import product
from dimod import BinaryQuadraticModel

Q = defaultdict(int)
for i, j in graph.edges:
    Q[(i,i)]+= -1
    Q[(j,j)]+= -1
    Q[(i,j)]+= 2

qubo = BinaryQuadraticModel(Q, "BINARY")
n = qubo.num_variables

best_energy = inf
best_state = []
for state in product([0, 1], repeat=n):
    x = np.array(state)
    energy = qubo.energy(x)
```

```
    if energy < best_energy:
        best_energy = energy
        best_state = state


# Interpretacja wyników
S0 = [node for node in graph.nodes if not best_state[node]]
S1 = [node for node in graph.nodes if best_state[node]]


uncut_edges = [(u, v) for u, v in graph.edges if best_state[u]==best_state[v]]

# narysowanie najlepszego znalezionego wyniku
nx.draw(graph, node_color=["r" if i in S0 else "c" for i in list(graph.nodes)],␣
 ↪with_labels=True,
      edge_color=["black" if e in uncut_edges else "lightgray" for e in␣
 ↪list(graph.edges)])
```



1.2 Problem komiwojażera (TSP) i jego sprowadzenie do postaci QUBO

1.2.1 1. Definicja problemu

Dany jest graf $G = (V, E)$, w którym każdej krawędzi (u, v) przypisana jest waga W_{uv}. Problem komiwojażera polega na znalezieniu takiego **cyklu Hamiltona**, który przechodzi przez każde miasto dokładnie raz i ma **minimalną sumę wag** użytych krawędzi.

1.2.2 2. Kodowanie binarne

Niech

$$N = |V|.$$

Wprowadzamy zmienne binarne

$$x_{v,j} \in \{0, 1\},$$

gdzie

$$x_{v,j} = 1 \quad \Longleftrightarrow \quad \text{miasto } v \text{ znajduje się na pozycji } j \text{ w trasie.}$$

Interpretacja jest więc następująca:

- indeks v oznacza miasto,
- indeks j oznacza pozycję miasta w cyklu.

W podstawowej wersji potrzeba N^2 zmiennych binarnych. Ponieważ w cyklu można bez straty ogólności ustalić jedno miasto jako pierwsze, liczbę zmiennych można zredukować do $(N-1)^2$.

1.2.3 3. Ograniczenia poprawnej trasy

Aby zakodowana konfiguracja odpowiadała poprawnej trasie, muszą być spełnione trzy warunki.

I Każde miasto występuje dokładnie raz

$$\sum_{j=1}^{N} x_{v,j} = 1 \qquad \forall v \in V.$$

II Na każdej pozycji znajduje się dokładnie jedno miasto

$$\sum_{v \in V} x_{v,j} = 1 \qquad \forall j = 1, \ldots, N.$$

III Kolejne miasta w trasie muszą być połączone krawędzią Jeżeli miasto u występuje na pozycji j, a miasto v na pozycji $j+1$, to przejście (u, v) musi należeć do zbioru krawędzi E.

Dla cyklu indeks $j+1$ rozumiemy modulo N, czyli po pozycji N wracamy do pozycji 1.

$$\sum_{(u,v) \notin E} \sum_{j=1}^{N} x_{u,j} x_{v,j+1} = 0$$

1.2.4 4. Hamiltonian dla cyklu Hamiltona

Hamiltonian dla problemu cyklu Hamiltona:

$$H_A = A\sum_{v=1}^{N}\left(1-\sum_{j=1}^{N}x_{v,j}\right)^2 + A\sum_{j=1}^{N}\left(1-\sum_{v=1}^{N}x_{v,j}\right)^2 + A\sum_{(u,v)\notin E}\sum_{j=1}^{N}x_{u,j}x_{v,j+1}.$$

Znaczenie poszczególnych składników:

1. pierwszy składnik wymusza, aby każde miasto pojawiło się dokładnie raz,
2. drugi składnik wymusza, aby każda pozycja była obsadzona dokładnie jednym miastem,
3. trzeci składnik karze przejścia między miastami, dla których nie istnieje krawędź w grafie.

Stała $A > 0$ jest parametrem kary.

1.2.5 5. Dodanie kosztu podróży

Dodajemy do H_A składnik kosztowy:

$$H_B = B\sum_{(u,v)\in E}W_{uv}\sum_{j=1}^{N}x_{u,j}x_{v,j+1}.$$

Całkowity Hamiltonian ma postać

$$H = H_A + H_B.$$

Składnik H_B nie wymusza poprawności trasy, lecz spośród wszystkich poprawnych tras wybiera tę o najmniejszym koszcie.

Aby nigdy nie opłacało się naruszać ograniczeń tylko po to, by obniżyć koszt, dobiera się parametry tak, by zachodziło

$$0 < B\max_{(u,v)\in E}W_{uv} < A.$$

Zakłada się też, że

$$W_{uv} \geq 0.$$

1.3 Główne nieefektywności tego przedstawienia TSP jako QUBO

1.3.1 1. Rozdmuchanie liczby zmiennych

W tej formulacji zmienna binarna $x_{v,j}$ oznacza, że miasto v jest odwiedzane na pozycji j. To daje N^2 bitów, a po unieruchomieniu jednego miasta nadal $(N-1)^2$.

Nie jest to jeszcze katastrofa, wiele klasycznych formulacji TSP też ma $O(N^2)$ zmiennych, ale w QUBO te zmienne tworzą zwykle **gęstsze sprzężenia** niż inne metody (np. MILP)

1.3.2 2. Zamiana prostych ograniczeń na kary kwadratowe pogarsza geometrię problemu

W QUBO warunki „każde miasto dokładnie raz" i „na każdej pozycji dokładnie jedno miasto" nie są zapisane jako twarde ograniczenia, tylko jako **kary kwadratowe**:

$$A\left(1-\sum_j x_{v,j}\right)^2, \qquad A\left(1-\sum_v x_{v,j}\right)^2.$$

Skutek jest taki, że:

- solver QUBO nie „wie", co jest ograniczeniem, a co celem - widzi tylko jedną energię
- trzeba ręcznie stroić współczynniki kar
- przestrzeń stanów zawiera ogromną liczbę **rozwiązań niepoprawnych**,

1.3.3 3. Konieczność separacji skal $A \gg B$ pogarsza uwarunkowanie modelu

W TSP trzeba dobrać B „na tyle małe", aby nigdy nie opłacało się łamać ograniczeń z H_A. Przykładowo:

$$0 < B \max(W_{uv}) < A.$$

To jest źródło nieefektywności, bo:

- jeśli A jest za małe, optimum może odpowiadać trasie niepoprawnej
- jeśli A jest za duże, funkcja celu jest zdominowana przez kary i robi się źle wyskalowana
- dla metod heurystycznych, j powstaje trudny kompromis między „feasibility" a „quality".

1.3.4 4. QUBO dla TSP jest zazwyczaj bardzo gęste

W tej formulacji wiele par zmiennych jest połączonych:

- zmienne dotyczące tego samego miasta na różnych pozycjach
- zmienne dotyczące różnych miast na tej samej pozycji
- zmienne opisujące kolejne pozycje w trasie
- przejścia zakazane

To znaczy, że macierz QUBO staje się bardzo gęsta. Dla praktycznego osadzania na sprzęcie problematyczne są:

- zbyt duża liczba spinów,
- duże separacje skal energii,

1.3.5 5. Symetrie i degeneracje zwiększają redundancję przestrzeni stanów

Można usunąć część redundancji, unieruchamiając jedno miasto jako pierwsze, co zmniejsza liczbę bitów z N^2 do $(N-1)^2$.

To jednak tylko częściowo pomaga. Nadal pozostają typowe symetrie:

- odwrócenie cyklu w przypadku grafu nieskierowanego,
- równoważne reprezentacje tej samej trasy,
- wiele stanów bliskich poprawności, ale nadal niepoprawnych.

1.4 Podsumowanie

Nieefektywność tej formulacji wynika głównie z tego, że **QUBO zamienia dobrze ustrukturyzowany problem z twardymi ograniczeniami na duży, gęsty i źle wyskalowany problem bez ograniczeń, w którym solver musi sam „odkrywać", co jest trasą poprawną, a co tylko niskoenergetycznym artefaktem**.

04_Algorytm_wyczerpującego_przeszukiwania

May 26, 2026

1 Algorytm wyczerpującego przeszukiwania

Znany także jako brute force. Polega na naiwnym policzeniu energi dla każdej konfiguracji i wybraniu optymalnej.

W kontekście rozwiązywania modelu Isinga/QUBO, warto wykorzystać fakt, że każda liczba naturalna posiada unikatową reprezentację binarną. Przykładowo:

$$123 = (1111011)_2$$

Zauważmy, że dowolny ciąg binarny długości N reprezentuje stan Isinga dla N spinów. Wtedy iterując po wszystkich liczbach od 0 do $2^N - 1$ mamy pewność, że sprawdzilismy wszystkie stany.

1.1 Implementacja naiwna

[1]:
```python
# Ustawienie ścierzek dla plików pomocniczych
from setup import setup_paths
setup_paths()

# Implementacja naiwnej pętli
import numpy as np
from math import inf
from tqdm import tqdm
from itertools import product
from funkcje_pomocnicze import calculate_energy

def brute_force_naive_qubo(Q: np.ndarray):
    best_energy = inf
    best_state = None
    n = Q.shape[0]

    def int_to_bits(n: int, width: int, dtype=np.uint8):
        shifts = np.arange(width - 1, -1, -1, dtype=np.uint64)
        return ((np.uint64(n) >> shifts) & 1).astype(dtype)

    for i in tqdm(range(2**n), desc="Wyczerpujące przeszukiwanie"):
        state = int_to_bits(i, n)
        energy = state @ Q @ state.T
```

```
        if energy < best_energy:
            best_energy = energy
            best_state = state

    return best_state, best_energy


# używając itertools
def brute_force_naive(J, h):
    best_energy = inf
    best_state = None
    n = len(h)

    for state in tqdm(product([-1, 1], repeat=n), desc="Wyczerpujące␣
 ↪przeszukiwanie", total=2**n):
        state = np.array(state)
        energy = calculate_energy(J, h, state, convention="dwave")
        if energy < best_energy:
            best_energy = energy
            best_state = state

    return best_state, best_energy
```

[2]:
```
# Instancja testowa 8 spinów

import time
from funkcje_pomocnicze import ising_to_qubo, read_instance, K8

J, h = read_instance(K8.path, convention="dwave")
Q, offset = ising_to_qubo(J, h)  # instancje testowe są zapisane jako ising

start = time.time()
state, energy = brute_force_naive_qubo(Q)
end = time.time()

print(f"Otrzymana energia: {energy + offset}")
print(f"Stan podstawowy: {K8.best_energy}")
print(f"czas: {end - start}")
```

```
Wyczerpujące przeszukiwanie: 100%|      | 256/256 [00:00<00:00,
85400.61it/s]

Otrzymana energia: -9.5
Stan podstawowy: -9.5
czas: 0.06356239318847656
```

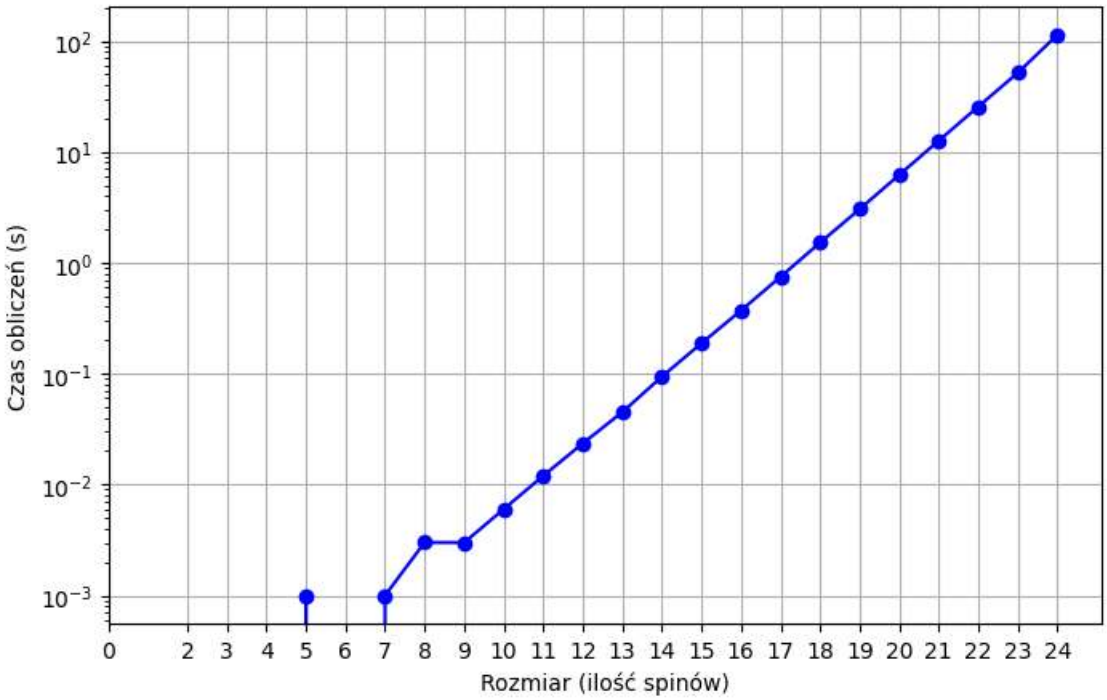
[3]:
```
# Instancja testowa 8 spinów

import time
from funkcje_pomocnicze import read_instance, K8
J, h = read_instance(K8.path, convention="dwave")  # pamiętamy o odpowiedniej␣
 ↪konwencji zapisu instancji

start = time.time()
state, energy = brute_force_naive(J, h)
end = time.time()

print(f"Otrzymana energia: {energy}")
print(f"Stan podstawowy: {K8.best_energy}")
print(f"czas: {end - start}")
```

```
Wyczerpujące przeszukiwanie: 100%|      | 256/256 [00:00<00:00,
128039.81it/s]

Otrzymana energia: -9.5
Stan podstawowy: -9.5
czas: 0.0039975643157958984
```

[4]:
```
# Polecam zobaczyć jak rozmiar n wpływa na czas obliczeń, dla n>20 czas␣
 ↪obliczeń zaczyna rosnąć bardzo szybko
import os
import time
import matplotlib.pyplot as plt
from IPython.utils.io import capture_output

times = []
is_automated_run = "PYTEST_CURRENT_TEST" in os.environ
n_range = list(range(2, 18 if is_automated_run else 25))
for n in tqdm(n_range, desc="Wyczerpujące przeszukiwanie dla różnych n (może␣
 ↪trwać kilka minut)"):

    J = np.triu(np.random.uniform(-1, 1, size=(n, n)), k=1)  # losowa gęsta␣
 ↪macierz górnotrójkątna
    h = np.random.uniform(-1, 1, size=n)  # losowy wektor
    start = time.time()
    with capture_output() as captured:
        state, energy = brute_force_naive(J, h)
    end = time.time()
    elapsed = end - start
    times.append(elapsed)

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

```
plt.plot(n_range, times, marker='o', linestyle='-', color='blue',␣
 ↪label='Execution Time')
plt.xlabel('Rozmiar (ilość spinów)')
plt.xticks([0] + n_range)
plt.yscale('log', base=10)
plt.ylabel('Czas obliczeń (s)')
plt.grid(True)
plt.show()
```

```
Wyczerpujące przeszukiwanie dla różnych n (może trwać kilka minut):
100%|      | 23/23 [03:35<00:00,  9.38s/it]
```



Złożonośc obliczeniowa tego algorytmu rośnie wykładniczo.

1.2 Efektywne implementacje

Istnieją dużo efektywniejsze implementacje tego algorytmu wykorzystujące GPU. Przykłaem jest oprogramowanie Omnisolver-bruteforce. Poniżej jest link do dokumentu pokazującego wyniki benchmarków.

wykres

2 Bibliografia

- Jałowiecki, K., Rams, M. M., & Gardas, B. (2021). Brute-forcing spin-glass problems with CUDA. *Computer Physics Communications*, *260*, 107728. DOI: 10.1016/j.cpc.2020.107728
- Jałowiecki, K., & Pawela, Ł. (2023). Omnisolver: An extensible interface to Ising spin–glass and QUBO solvers. *SoftwareX*, *24*, 101559. DOI: 10.1016/j.softx.2023.101559

05_Przegląd_algorytmow_hurystycznych

May 26, 2026

1 Gwarancja optymalności

Wyczerpujące przeszukiwanie jest jednym z niewielu algorytmów które **gwarantują** znalezienie stanu podstawowego. Przykładowe inne metody z taką samą gwarancją to:

- Branch-and-Bound
- Branch-and-cut
- Sieci tensorowe
- "idealne" wyżarzanie kwantowe

Każdy z w/w algorytmów skaluje się wykładniczo. Często gwarancja optymalość jest tylko przy zbierzności w nieskończoności. Z tego powodu najczęściej stosuje się algorytmy *heurystyczne* tzn. takie, które potrafią znaleść dobre rozwiązanie, ale nie gwarantują że będzie ono najlepsze możliwe (optymalne).

2 Przegląd algorytmów heurystycznych inspirowanych fizycznie

Szczegóły implementacyjne zostaną omówione drugiego dnia

- symulowanie wyżarzanie
- wyżarzanie równoległe
- symulowana bifurkacja

Kwantowy model Isinga

wprowadzenie

Do matematycznego opisu kwantowego modelu potrzebujemy *macierzy Pauliego*. Są to macierze 2×2 które opisują interakcje spinu z polem magnetycznym. Konkrektnie, będziemy potrzebować macierzy σ^z oraz σ^x zdefiniowanych jako:

$$ \sigma^z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \sigma^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} $$

Ponadto będziemy potrzebować wektorów i własności własnych. Dla przypomnienia, jeżeli A jest macierzą, to wektor v nazywamy **wektorem własnym** jeżeli spełniony jest warunek:

$$ Av = \lambda v $$

gdzie λ jest niezerowym skalarem nazywanym **wartością własną**.

Definicja

Kwantowy model Isinga jest definiowany w następujący sposób:

$$ \mathcal{H} = \sum_{(i, j)} J_{ij} \sigma_i^z \sigma_j^z + \sum_{i=1}^{N} h_i \sigma_i^z $$

Warto pamiętać, że w przypadku działania macierzy Pauliego σ^{α} ($\alpha = x,z$) na spin i, zapis σ^{α}_i oznacza

\begin{equation*} \sigma^{\alpha}_i = I \otimes I \otimes \ldots \otimes I \otimes \sigma^{\alpha} \otimes I \otimes \ldots \otimes I \end{equation*}

gdzie $\otimes$ jest iloczynem tensorowym (lub iloczynem Kroneckera), z i-tym czynnikiem σ^{α}.

Iloczyn tensorowy jest definiowany następująco: $$ A \otimes B \;=\; \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B\\ a_{21}B & a_{22}B & \cdots & a_{2n}B\\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}. $$ Przykład dla macierzy 2x2

$$ A=\begin{bmatrix} a_{11} & a_{12}\\ a_{21} & a_{22} \end{bmatrix},\quad B=\begin{bmatrix} b_{11} & b_{12}\\ b_{21} & b_{22} \end{bmatrix}, \quad A\otimes B= \begin{bmatrix} a_{11}B & a_{12}B\\ a_{21}B & a_{22}B \end{bmatrix} = \begin{bmatrix}

a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12}\\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22}\\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12}\\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix} $$

Iloczyn tensorowy jest operacją łączną, więc mozna go liczyc niejako "rekurencyjnie":

$$ A\otimes B \otimes C = (A\otimes B)\otimes C = A\otimes (B \otimes C) = D \otimes C = A \otimes E = F $$ gdzie $D = A\otimes B$, $E = B \otimes C$ oraz $F = A\otimes B \otimes C$

Związek między klasycznym a kwantowym modelem Isinga

Hamiltonian $\mathcal{H}$ jest olbrzymią macierzą o rozmiarze $2^n \times 2^n$, gdzie n jest ilością spinów, natomiast klasyczny H, to jedynie pewna suma wskazująca wartość energii dla danego stanu. Jak, tak różne obiekty (liczba i macierz) mają się do siebie? Odpowiedź leży w specyfice mechaniki kwantowej.

W pewnym uproszczeniu, jeżeli Hamiltonian opisujący układ kwantowy jest macierzą, to wektory własne tej macierzy opisują stany, które ten układ może przyjąć a odpowiadające im własności własne są energiami tych stanów.

Przyjmijmy notację: $$ \ket{\uparrow} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \ket{\downarrow} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} $$

Można pokazać, że wektory własne macierzy σ^z to właśnie $\ket{\uparrow}$ i $\ket{\downarrow}$ z odpowiadającymi im wartościami własnymi $+1$ i -1. Zatem macierz σ^z opisuje pojedyńczy spin.

W ten sam sposób, wektory i własności własne kwantowego modelu Isinga bez pola poprzecznego odpowiadają stanom i energią w klasycznym modelu Isinga.

In [1]:
```
# Oba modele Isinga - klasyczny i kwantowy dają te same stany
# pokażemy to wprost na bardzo małym przykładzie (na razie ignorując pole po

import numpy as np
import qutip as qt
from itertools import product

J = {(1, 2): 0.5, (1, 3): -1, (2, 3): 0.75}
h = {1: 1, 2: -1, 3: 0.5}

# kwantowy
sigma_1 = qt.tensor([qt.sigmaz(), qt.qeye(2), qt.qeye(2)])
sigma_2 = qt.tensor([qt.qeye(2), qt.sigmaz(), qt.qeye(2)])
sigma_3 = qt.tensor([qt.qeye(2), qt.qeye(2), qt.sigmaz()])

H_quantum = J[(1, 2)] * sigma_1 * sigma_2 + J[(1, 3)] * sigma_1 * sigma_3 +
            h[1] * sigma_1 + h[2] * sigma_2 + h[3] * sigma_3
```

```
eigenvalues, eigenstates = H_quantum.eigenstates()

def decode_eigenstate(eigenstate):
    state = eigenstate.full()
    idx, _ = np.nonzero(state)
    idx = idx.item()
    basis_states = list(product([1,-1], repeat=3))
    return basis_states[idx]

# klasyczny
energies_states = {}

for state in product([-1,1], repeat=3):
    energy = sum([state[i-1] * state[j-1] * J[(i, j)] for  (i, j) in J.keys(

    energies_states[energy] = state

energies_states = dict(sorted(energies_states.items()))

# podsumowanie
print("Wartości własne macierzy H:", eigenvalues)
print("Wszystkie energie klasycznego modelu:", list(energies_states.keys()))
print("------------")
print("Stan odpowiadający najmiejszej wartości własnej", decode_eigenstate(e
print("Stan podstawowy klasycznego modelu: ", energies_states[-4.75])
```

```
Wartości własne macierzy H: [-4.75 -0.25 -0.25  0.25  0.25  0.75  1.25  2.7
5]
Wszystkie energie klasycznego modelu: [-4.75, -0.25, 0.25, 0.75, 1.25, 2.75]
------------
Stan odpowiadający najmiejszej wartości własnej (-1, 1, -1)
Stan podstawowy klasycznego modelu:  (-1, 1, -1)
```

Kwantowy model Isinga z polem poprzecznym

Na razie kwantowy model Isinga jest tylko wyłącznie innym matematycznym opisem tego samego obiektu.

Kwantowy model Isinga z polem poprzecznym jest definiowany w następujący sposób:

$$ \mathcal{H} = \Gamma \sum_{i=1}^N \sigma^x_i + \sum_{(i, j)} J_{ij} \sigma_i^z \sigma_j^z + \sum_{i=1}^{N} h_i \sigma_i^z $$

gdzie Γ jest parametrem określającym siłę pola poprzecznego.

Twierdzenie adiabatyczne

Dla potrzeb kursu, sformułujemy intuicyjnie twierdzenie adiabatyczne. Pełne i formalne zapisy mechaniki kwantowej pominiemy, zostawiając je do dogłębnej analizy zainteresowanym kursantom.

Twierdzenie adiabatyczne (w mechanice kwantowej) Rozważmy system kwantowy zmieniający się w czasie. Jeżeli na początku ewolucji znajduje się on w stanie podstawowym, to przy pewnych technicznych założeniach, o ile zmiany systemu następują **wystarczająco wolno**, uzyskany na końcu system kwantowy jest również w stanie podstawowym. W skrócie, **powolna** ewolucja systemu przekształca stan podstawowy na stan podstawowy.

Sformułowanie wyżarzania kwantowego

W wyżarzaniu kwantowym wykorzystuje się to twierdzenie, aby znaleźć stan podstawowy (minimum globalne) hamiltonianu Isinga, który koduje problem optymalizacyjny. Proces wygląda tak:

1. Startujemy od prostego hamiltonianu $H_{initial}$, którego stan podstawowy jest znany (np. prosta superpozycja).
2. Stopniowo przekształcamy go w hamiltonian problemu $H_{problem}$, którego minimum chcemy znaleźć.
3. Jeśli proces jest adiabatyczny (czyli wystarczająco powolny), to układ przejdzie z podstawowego stanu $H_{initial}$ do podstawowego stanu $H_{problem}$, dając rozwiązanie.

tutaj wyżarzanie kwantowe zostanie zaimplementowane następująco:

$$ \mathcal{H}(s) = A(s)H_{initial} + B(s)H_{Ising} =A(s) \sum_{i}^{N} \sigma_i^x + B(s) \left(\sum_{j < i} J_{ij} \sigma_i^z \sigma_j^z + \sum_{i=1}^{N} h_i \sigma_i^z \right) $$

gdzie $s = t/\tau$, τ to całkowity czas wyżarzania a funkcje $A(s)$ i $B(s)$ to dowolne ciągłe funkcje spełniające warunki $A(0) \gg B(0)$, oraz $B(1) \gg A(1)$ oraz $A(1) \approx 0$.

Ewolucja układu

Ewolucja układu jest opisywana przez równanie Schrödingera

$$ i \frac{\partial}{\partial t} \ket{\Psi (t)} = \mathcal{H}\ket{\Psi (t)} $$

gdzie t jest czasem, $\ket{\Psi (t)}$ jest wektorem stanu a $\mathcal{H}$ jest Hamiltonianem opisującym układ.

Jako że mechanika kwantowa jest z natury probabilistyczna i na końcu ewolucji otrzymujemy pewien wektor prawdopodobieństw, interesuje nas wartość oczekiwana pomiaru energi $<\mathcal{H}(t)>$. Jest to średnia ważona wszystkich możliwych pomiarów klasycznego stanu (tj. wektoru spinów).

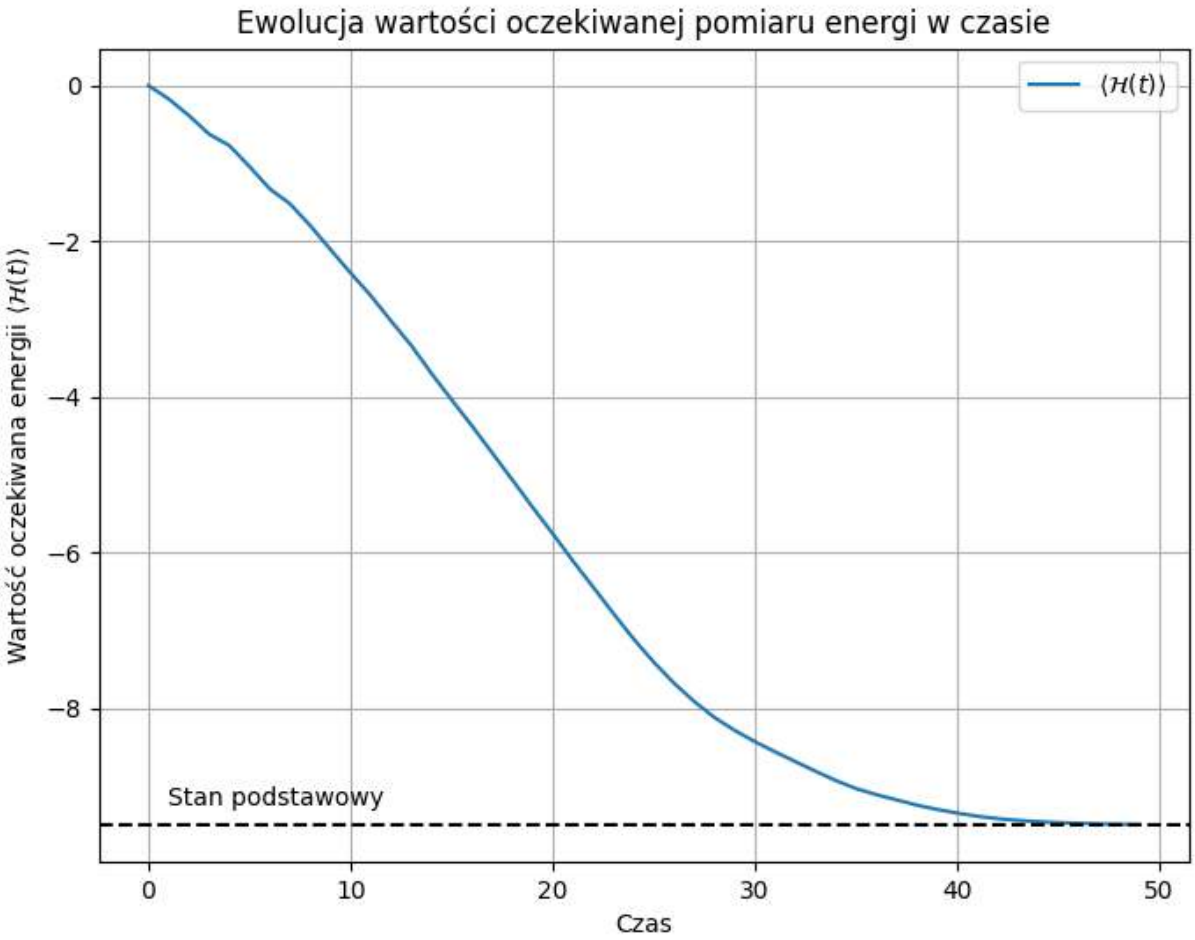
In [1]:
```python
# Funkcje pomocniczne które będziemy używać do robienia symulacji wyżarzania
from setup import setup_paths
setup_paths()

import os

import qutip as qt
import pandas as pd
import numpy as np

from typing import Optional
from math import pi
from qutip.measurement import measure, measurement_statistics


def create_spin_operators(N: int):
    #  Konstruujemy odpowiednie operatory działające na odpowiednim qubicie
    sx_list = []
    sz_list = []
    for i in range(N):
        # sigma_x/z na i-tym miejscu, na reszcze identyczności
        op_list_x = [qt.sigmax() if k==i else qt.qeye(2) for k in range(N)]
        op_list_z = [qt.sigmaz() if k==i else qt.qeye(2) for k in range(N)]
        # z listy tworzymy jedna dużą macierz (operator) przez użycie iloczy
        sx_list.append(qt.tensor(op_list_x))
        sz_list.append(qt.tensor(op_list_z))
    return sx_list, sz_list


def create_problem_hamiltonian(N: int, sz_list: list, J: dict, h: Optional[d
    #  Zakładamy że qubity są ponumerowane od 0 do N-1
    linear = sum([h[i] * sz_list[i] for i in range(N)]) if h else 0
    H_problem = linear + sum([J[(i, j)] * sz_list[i] * sz_list[j] for (i, j)
    return H_problem


def load_dwave_schedule(path: os.PathLike):
    df = pd.read_excel(path, usecols=["s", "A(s) (GHz)", "B(s) (GHz)"])
    # Przejście z GHz na naturalne jednostki oraz aplikacja skalaru 1/2
    df["A(s)"] = df["A(s) (GHz)"].map(lambda x: x*pi)
    df["B(s)"] = df["B(s) (GHz)"].map(lambda x: x*pi)
    return df
```

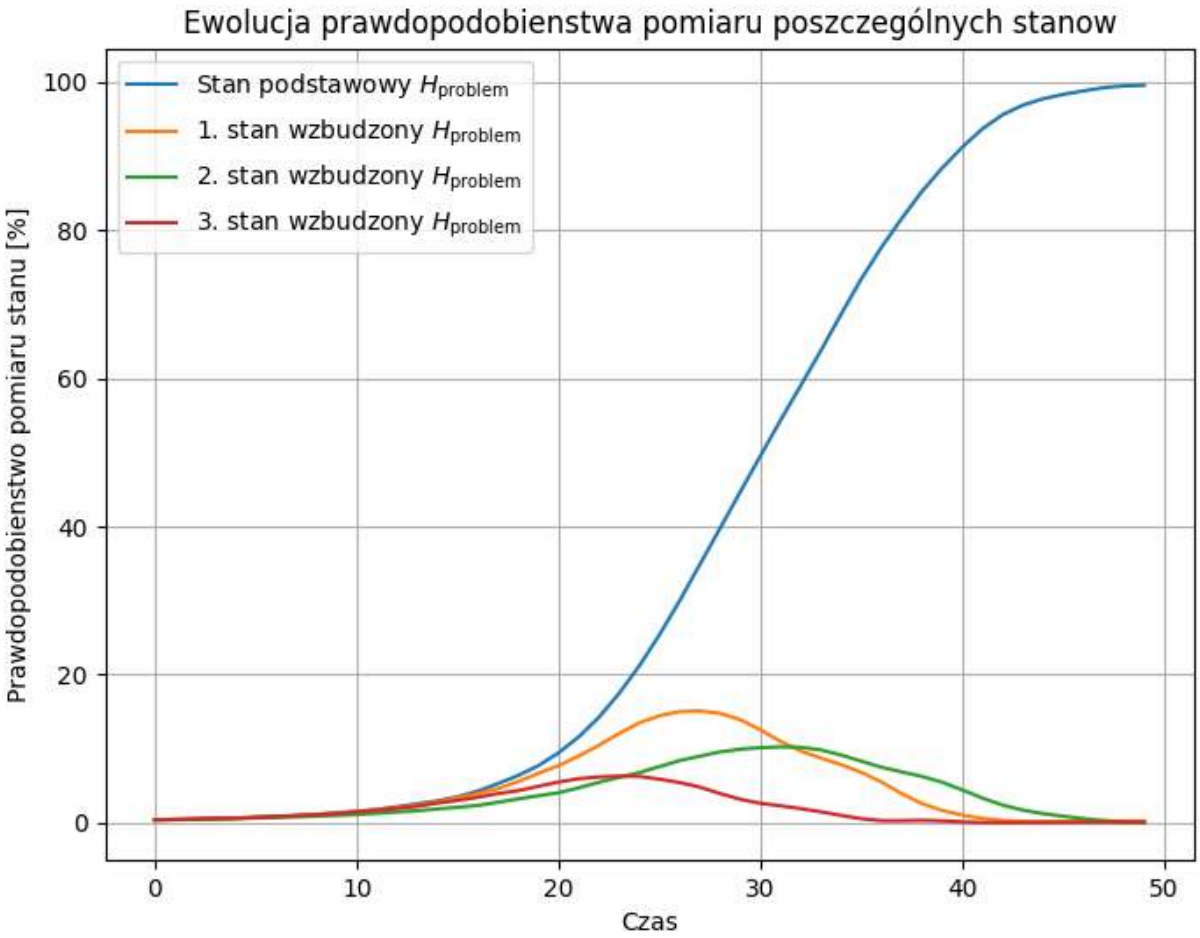
```
# Kod do symulacji wyżarzania kwantowego. W tej symulacji ewoluujemy równani
def create_full_hamiltonian(J: dict, h: dict,  time: int, A: Optional[callab
    if A is None:
        def A(t, T):
            return 1 - t/T

    if B is None:
        def B(t, T):
            return t/T

    N = len(h)

    # Tworzymy odpowiednie operatory sigma_x i sigma_z
    sx_list, sz_list = create_spin_operators(N)

    # Tworzymy  Hamiltonian początkowy H_initial
    H_initial =  -1 * sum(sx_list)

    # Tworzymy hamiltonian probelmu
    H_problem = create_problem_hamiltonian(N, sz_list, J, h)

    # Pełen hamiltonian H(t)
    # H(t) = A(t)*H_initial + B(t)*H_problem
    H_t = qt.QobjEvo([[H_initial, A], [H_problem, B]], args={"T": time})

    # Przygotuj stan podstawowy dla początkowego hamiltonianiu
    psi0 = H_initial.groundstate()[1]

    # Do symulacji potrzebujemy H_t i psi0, H_problem potrzebujemy do wykona
    return H_t, psi0, H_problem


def simulate(hamiltonian, psi0, H_problem, time: int, num_steps: int):

    t_list = np.linspace(0, time, num_steps, endpoint=True)
    result = qt.sesolve(hamiltonian, psi0, t_list, e_ops=H_problem, options=
    states = result.states
    expectatnions = result.expect

    return states, expectatnions
```

In [2]:

```
# Ewolucja wartości oczekiwanej w czasie
# Ewolucja prawdopodobienstwa pomiaru stanu w czasie

import matplotlib.pyplot as plt

from qutip.measurement import measure, measurement_statistics
from funkcje_pomocnicze import read_instance_dict, K8


J, h = read_instance_dict(K8.path)

# Czas wyżarzania
T = 50
```

```
num_steps = 50

H_t, psi0, H_problem = create_full_hamiltonian(J, h, T)


states, expect = simulate(H_t, psi0, H_problem, T, num_steps)
final_state = states[-1]

# Obrazek wartość oczekiwana
x = range(num_steps)
y1 = expect[0]


plt.figure(figsize=(8, 6))

plt.plot(x, y1, label=r"$\langle \mathcal{H}(t) \rangle$")

plt.axhline(y=-9.5, color='black', linestyle='--')

plt.title("Ewolucja wartości oczekiwanej pomiaru energi w czasie")
plt.xlabel("Czas")
plt.ylabel(r"Wartość oczekiwana energii $\langle \mathcal{H}(t) \rangle$")

plt.text(1, -9.25, "Stan podstawowy", fontsize=10, color='black')
plt.grid(True)
plt.legend()
plt.show()

y2_1 = [measurement_statistics(state, H_problem)[2] for state in states]
y2_2 = [data[:4] for data in y2_1]
y2 = [[x * 100 for x in data] for data in y2_2]

plt.figure(figsize=(8, 6))
plt.plot(x, y2)
plt.title("Ewolucja prawdopodobienstwa pomiaru poszczególnych stanow")
plt.xlabel("Czas")
plt.ylabel("Prawdopodobienstwo pomiaru stanu [%]")
plt.grid(True)
plt.legend([r"Stan podstawowy $H_{\text{problem}}$", r"1. stan wzbudzony $H_
plt.show()
```


Ewolucja wartości oczekiwanej pomiaru energi w czasie
⟨ℋ(t)⟩
Wartość oczekiwana energii ⟨ℋ(t)⟩
0
−2
−4
−6
−8
Stan podstawowy
0
10
20
30
40
50
Czas



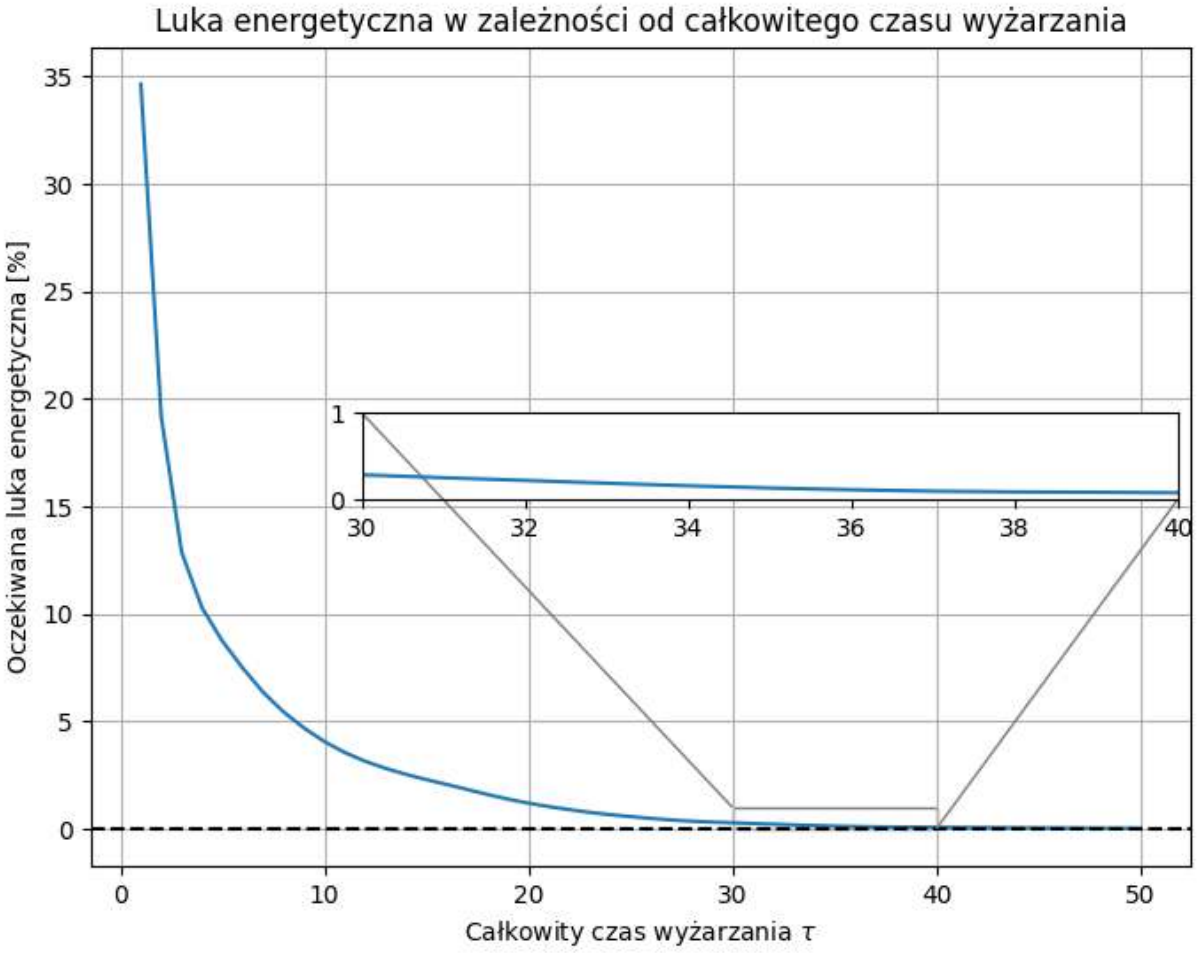
Tak jak się spodziewaliśmy, wraz z czasem prawdopodobienstwo pomiaru stanu podstawowego rośnie. Pomiar jest probabilistyczny, więc rozpatrujemy wartość oczekiwaną pomiaru.

Demonstracja adiabatyczności

Przeprowadzimy wiele symulacji, każdorazowo wykonując wyżarzanie kwantowe przez podany czas

In [3]:
```
# rysunek dla demonstracji adiabatyczności

import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance_dict, K8
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes, mark_ir


J, h = read_instance_dict(K8.path)
num_steps = 10
T = 50
gaps = []
for t in range(1, T+1):
    H_t, psi0, H_problem = create_full_hamiltonian(J, h, t)
    states, expect = simulate(H_t, psi0, H_problem, t, num_steps)
    energy = expect[0][-1]
```

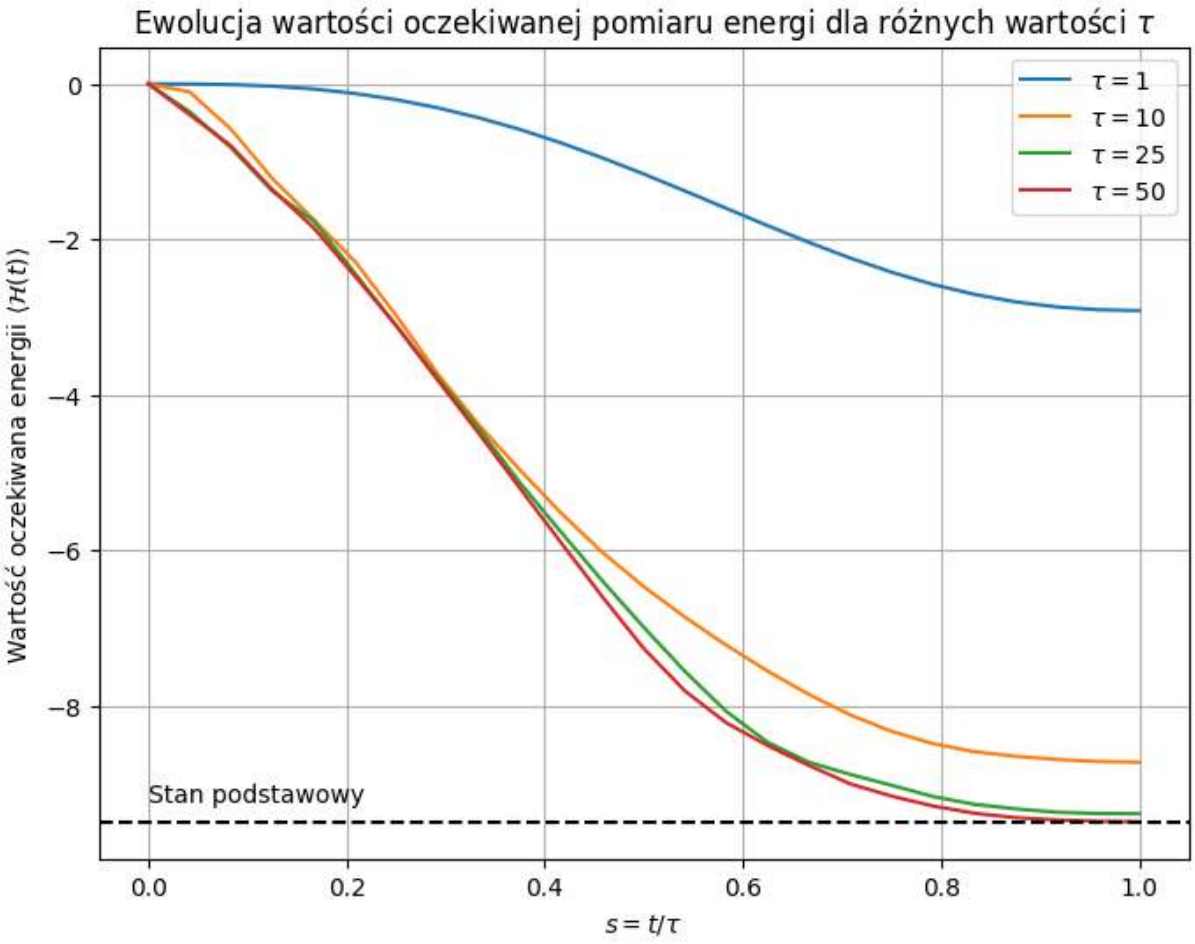
```
    gap = (K8.best_energy - energy)/(2*K8.best_energy) * 100
    gaps.append(gap)

x = range(1, T+1)
fig, ax = plt.subplots(figsize=(8, 6))

plt.plot(x, gaps)
plt.axhline(y=0, color='black', linestyle='--')

# Define the region of interest for the zoomed-in plot
x1, x2 = 30, 40  # x-limits for inset
# Compute corresponding y-limits for the sine function
y1, y2 = gaps[x1], gaps[x2]
# To have a little margin around the data points, adjust y-limits
delta_y = 0.1
y1 -= delta_y
y2 += delta_y

axins = zoomed_inset_axes(ax, zoom=4, loc="right")
axins.plot(x, gaps)

# Limit the view to the region of interest
axins.set_xlim(x1, x2)
axins.set_ylim(0, 1)



# Draw lines indicating the area of the inset on the main plot
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

ax.set_title("Luka energetyczna w zależności od całkowitego czasu wyżarzania
ax.set_xlabel(r"Całkowity czas wyżarzania $\tau$")
ax.set_ylabel("Oczekiwana luka energetyczna [%]")
ax.grid(True)

plt.show()
```



In [4]:

```python
# Demonstracja luki energetycznej w zależności od całkowitego czasu wyżarzan

import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance_dict, K8
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes, mark_ir


J, h = read_instance_dict(K8.path)
num_steps = 25
times = [1, 10, 25, 50]
energies = {}
for t in times:
    H_t, psi0, H_problem = create_full_hamiltonian(J, h, t)
    states, expect = simulate(H_t, psi0, H_problem, t, num_steps)
    energies[t] = expect

x = np.linspace(0, 1, num=num_steps, endpoint=True)
fig, ax = plt.subplots(figsize=(8, 6))

for t in times:
    plt.plot(x, energies[t][0], label=rf"$\tau = ${t}")

plt.text(0, -9.25, "Stan podstawowy", fontsize=10, color='black')
plt.axhline(y=-9.5, color='black', linestyle='--')
plt.legend()
plt.title(r"Ewolucja wartości oczekiwanej pomiaru energi dla różnych wartośc
```

```
plt.xlabel(r"$s = t / \tau$")
plt.ylabel(r"Wartość oczekiwana energii $\langle \mathcal{H}(t) \rangle$")
plt.grid()
plt.show()
```



Bibliografia

- Albash, T., and Lidar, D. A. (2018). Adiabatic quantum computation. *Reviews of Modern Physics, 90(1)*, 015002.
- Tanaka, S., Tamura, R., and Chakrabarti, B. K. (2017). *Quantum spin glasses, annealing and computation.* Cambridge University Press.

08_D_Wave

May 26, 2026

1 Wyżarzacze kwantowe

Wyżarzaczem kwantowym nazywamy maszynę, która fizycznie implementuje algorytm wyżarzania kwantowego. Obecnie najpopularniejsze są urządzenia produkowane przez firmę DWave. W sercu każdej maszyny leży procesor kwantowy, *Quantum Proccesing Unit* (QPU), składający się z kubitów połączonych couplerami (czasami słowo *coupler* tłumaczy się jako "sprzęgacz"). W tym szkoleniu nie będziemy wdawać się w szczegóły jak te kubity i couplery fizycznie wyglądają. Strukturę połączeń między kubitami w QPU nazywamy **topologią**.

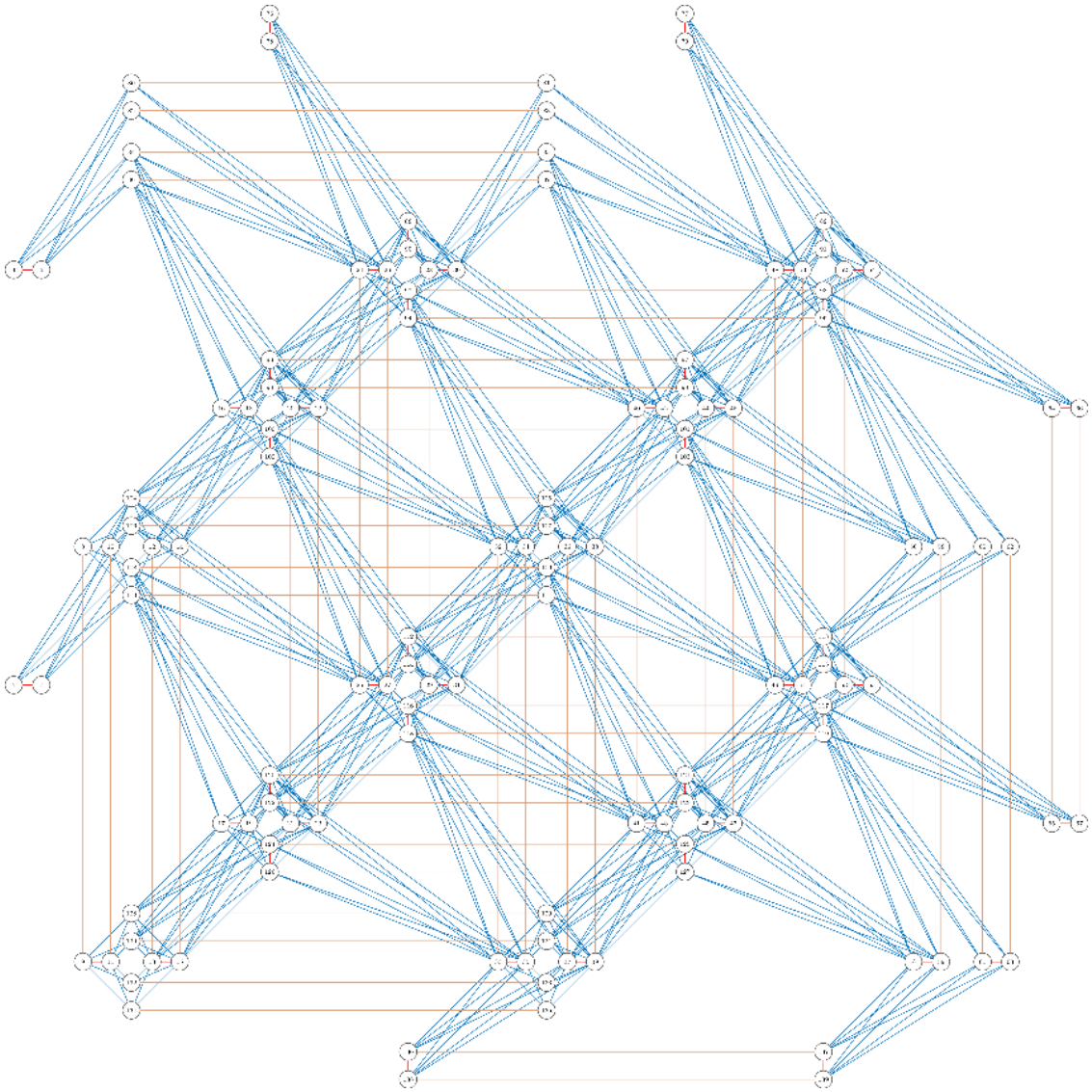
1.1 Topologie procesorów DWave

1.1.1 Chimera

Jest to pierwsza i najstarsza z istniejących topologii. Składa się z komórek jednostkowych (ang. *Unit Cell*) połączonych ze sobą w kształt kraty. Każda z komórek składa się z ośmiu kubitów połączonych ze sobą jak pełny graf dwudzielny $K_{4,4}$. Poniżej jest przedstawiony przykład Chimery **C2**

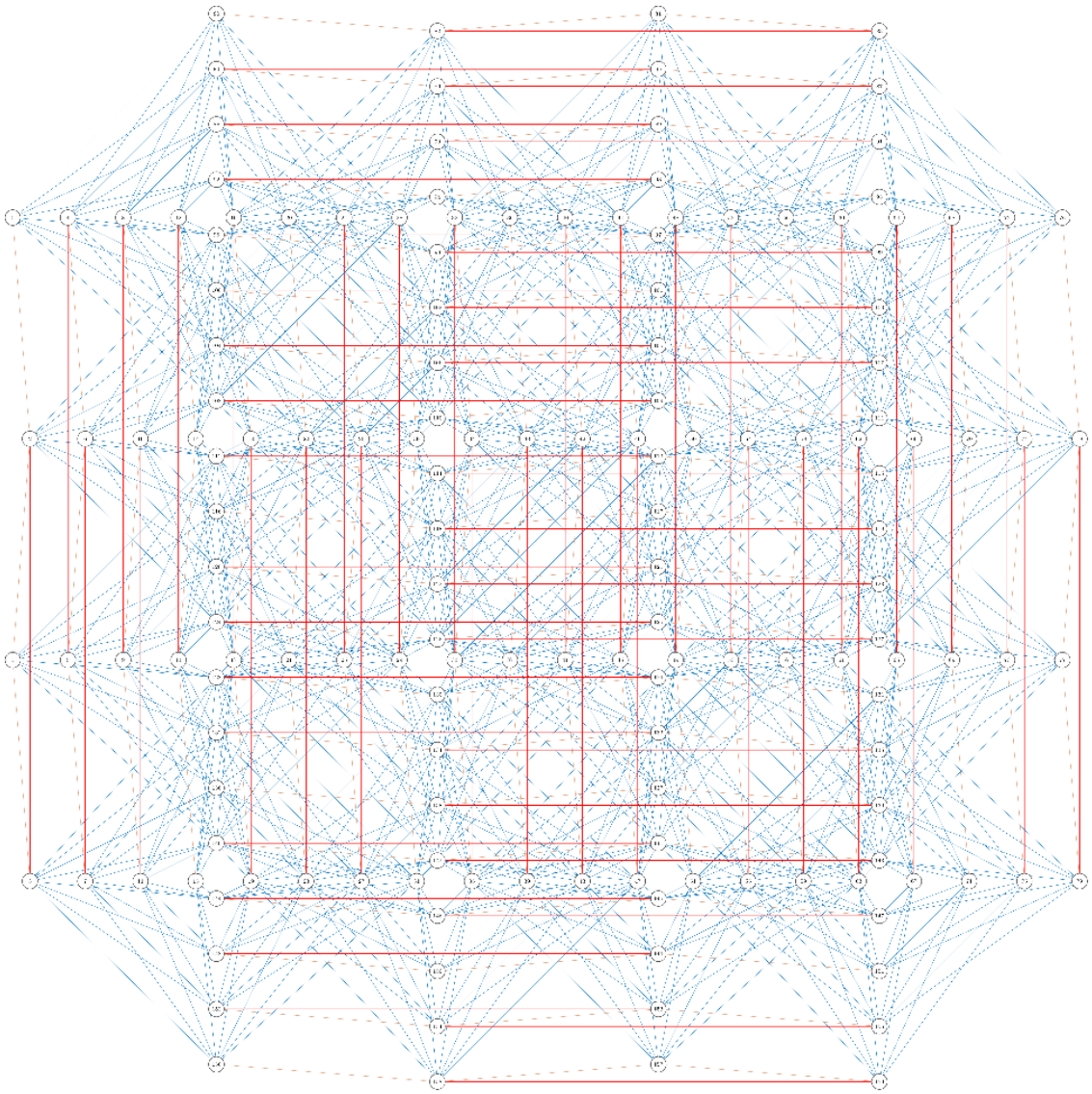
1.1.2 Pegasus

Jest to obecnie standardowa topologia używana w wyżarzaczach D-Wave. Nie ma najprostrzej konstrukcji.



1.1.3 Zephyr

Najnowocześniejsza topologia. Urządzenia które ją implementują są obecnie w fazie prototypów. **Z2** czyli zephyr złorzony z 2x2 komórek jednostkowych. Na pierwszy rzut oka widać zarówno więcej kubitów jak i znacznie gęściejszą siatkę połączeń.



1.2 Embedding

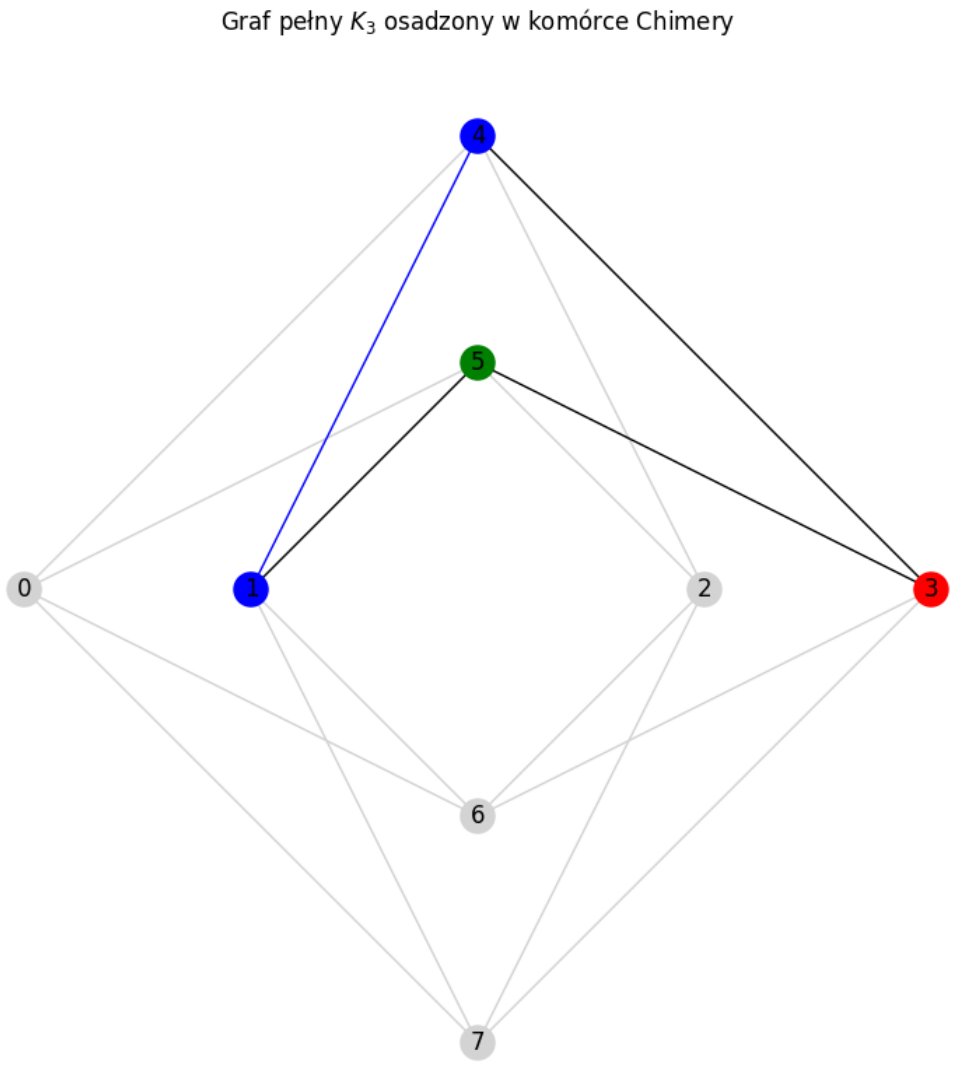
Z powodu tego jak jest skonstuowana sieć połączeń w QPU, wiele problemów trzeba osadzić. Powoduje to utworzenie tzw. *łańcuchów*. Oznacza to, że jeden qubit logiczny (zmienna) jest reprezentowany przez kilka qubitów fizycznych połączonych w łańcuch. Każdy z nich powinien mieć ten sam spin. Gdy tak się nie dzieje mówimy o *łamaniu łańcucha*.

[1]:
```
import networkx as nx
import dwave_networkx as dnx
import minorminer
import matplotlib.pyplot as plt


chimera = dnx.chimera_graph(1, 1, 4)  # pojedyńcza komórka chimery
K3 = nx.complete_graph(3)
```

```python
# Automatyczne szukanie osadzenia
embedding = minorminer.find_embedding(K3, chimera)
print("Znaleziono osadzenie:")
for k3_node, chain in embedding.items():
    if len(chain) > 1:
        print(f"  K3 wierzchołek {k3_node} -> łańcuch {chain}")
    else:
        print(f"  K3 wierzchołek {k3_node} -> wierzchołek {chain[0]}")



chain_colors = {k: color for k, color in zip(embedding.keys(), ["red", "green",␣
 ↪"blue", "purple"])}

# Budujemy mapę kolorów dla komórki chimery
node_color = {}
edge_color = {}

for k3_node, chain in embedding.items():
    for q in chain:
        node_color[q] = chain_colors[k3_node]
    if len(chain) > 1:
        v, w = chain
        edge_color[(v, w)] = chain_colors[k3_node]
        edge_color[(w, v)] = chain_colors[k3_node]

for (v, w) in K3.edges():
    x = embedding[v]
    y = embedding[w]
    if len(x) == len(y) == 1:
        edge_color[(x[0], y[0])] = "black"
        edge_color[(y[0], x[0])] = "black"
    elif len(x) > 1:
        edge_color[(x[0], y[0])] = "black"
        edge_color[(y[0], x[0])] = "black"
        edge_color[(x[1], y[0])] = "black"
        edge_color[(y[0], x[1])] = "black"
    else:
        edge_color[(x[0], y[0])] = "black"
        edge_color[(y[0], x[0])] = "black"
        edge_color[(x[0], y[1])] = "black"
        edge_color[(y[1], x[0])] = "black"

# TODO: napisz lepiej
```

```
# Kolor dla kubitów i krawędzi nie używanych w embeddingu
default_color = "lightgray"

node_colors = [node_color.get(node, default_color) for node in chimera.nodes()]
edge_colors = [edge_color.get(edge, default_color) for edge in chimera.edges()]


plt.figure(figsize=(8, 8))
dnx.draw_chimera(chimera, node_color=node_colors, edge_color=edge_colors,␣
 ↪with_labels=True, node_size=300)
plt.title(rf"Graf pełny $K_3$ osadzony w komórce Chimery")
plt.show()
```

```
Znaleziono osadzenie:
  K3 wierzchołek 0 -> wierzchołek 3
  K3 wierzchołek 1 -> wierzchołek 5
  K3 wierzchołek 2 -> łańcuch [1, 4]
```



w powyższym obrazku czarne krawędzie oznaczają krawędzie w oryginalnym grafiem a krawędzie w kolorze wierzchołków oznaczają łańcuch.

1.2.1 Większy przykład na prawdziwej architekturze

Najpierw trzeba klikną “Source - QPU”, a następnie po lewej stronie “show chains” oraz “show broken chains

[ ]:
```
# dokładnie co się dzieje w kodzie będzie opisane później
# niezbędy token by móc się połączyć
with open("token.txt", "r") as f:
    token = f.read()
```

```python
#Zdefiniowanie maszyny której będziemy używać
from dwave.system import DWaveSampler, AutoEmbeddingComposite
import dwave.inspector
import networkx as nx


sampler = DWaveSampler(token=token, solver="Advantage_system4.1")
graph = nx.complete_graph(50)
h = {node: 0.5 for node in graph.nodes}
J = {edge: 1 for edge in graph.edges}

# automatycznie mapujemy nasz problem na QPU
solver = AutoEmbeddingComposite(sampler)

# Rozwiązujemy problem
solution = solver.sample_ising(h, J)
dwave.inspector.show(solution)
```

09_zadania_dodatkowe

May 26, 2026

1 Zadania dodatkowe niewymagające programowania

1.1 1. Problem plecakowy

Na podstawie modelu przedstawionego w notatniku należy przekształcić końcową funkcję celu

$$A\left(\sum_{i=1}^{n} w_i x_i + \sum_{k=0}^{K} 2^k y_k - W\right)^2 - B\sum_{i=1}^{n} v_i x_i,$$

do **jawnej postaci QUBO**, tj. do postaci

$$\sum_i Q_{ii} z_i + \sum_{(i,j)} Q_{ij} z_i z_j.$$

W szczególności należy: - rozwinąć wyrażenie kwadratowe, - uporządkować wyrazy liniowe oraz dwuliniowe, - wyznaczyć współczynniki macierzy Q odpowiadające otrzymanemu modelowi.

1.2 2. Problem podziału grafu

Rozważmy graf prosty $G = (V, E)$ o parzystej liczbie wierzchołków

$$N = |V|.$$

Celem jest wyznaczenie takiego podziału zbioru wierzchołków V na dwa rozłączne podzbiory o jednakowej liczności, tj. po $\frac{N}{2}$ wierzchołków każdy, aby liczba krawędzi łączących oba podzbiory była minimalna.

Należy: - sformułować powyższy problem w postaci **modelu QUBO** lub **modelu Isinga**, - uzasadnić wybór dogodniejszej reprezentacji, - (opcjonalnie) sprowadzić otrzymany model do **postaci jawnej**.

1.3 3. Problem pełnego pokrycia

Rozważmy zbiór

$$U = \{1, \dots, n\},$$

oraz rodzinę jego podzbiorów

$$V_i \subseteq U, \qquad i = 1, \dots, N,$$

spełniającą warunek

$$U = \bigcup_{i=1}^{N} V_i.$$

Pytanie polega na rozstrzygnięciu, czy istnieje taki wybór podrodziny

$$R \subseteq \{V_1, \dots, V_N\},$$

że: 1. zbiory należące do R są parami rozłączne, tzn.

$$V_i \cap V_j = \emptyset \qquad \text{dla } i \neq j,$$

2. suma zbiorów należących do R pokrywa cały zbiór U, tj.

$$\bigcup_{V \in R} V = U.$$

Innymi słowy, należy ustalić, czy istnieje wybór rozłącznych podzbiorów, które łącznie tworzą dokładnie zbiór U.

Należy: - przedstawić ten problem w postaci **modelu QUBO** lub **modelu Isinga** oraz wskazać, która z tych reprezentacji jest bardziej naturalna w tym przypadku, - rozszeżyć ten problem do problemu optymalizacyjnego. Znaleźć najmniejszy zbiór R spełniający podane wymagania. - (opcjonalnie) sprowadzić model do **postaci jawnej**.

1.4 4. Problem Max-Cut

Należy sformułować problem **Max-Cut** (notatnik) w postaci **modelu Isinga**.

–

01_Praca_z_wyżarzaczami_kwantowymi

May 26, 2026

1 Ocean Software

Ocean Software to zestaw narzędzi programistycznych, umożliwiający modelowanie i rozwiązywanie problemów optymalizacyjnych przy użyciu kwantowych procesorów wyżarzania. Składa się z wielu powiązanych ze sobą paczek w pythonie które pozwalają modelować problemy optymalizacyjne, mapować je na odpowiednie formaty i przesyłać do rozwiązywania na komputerach kwantowych D-Wave.

[1]:
```
# niezbędy token by móc się połączyć
with open("token.txt", "r") as f:
    token = f.read()

#Zdefiniowanie maszyny której będziemy używać
from dwave.system import DWaveSampler

sampler = DWaveSampler(token=token)

# Parametry maszyny do której mamy dostęp
print("Parametry: ")
print("QPU: ", sampler.properties["chip_id"] )
print("Architektura: ", sampler.properties["topology"])
print("Ilość kubitów: ", sampler.properties["num_qubits"])
print("Ilość połączen: ", len(sampler.properties["couplers"]))
```

```
Parametry:
QPU:  Advantage_system4.1
Architektura:  {'type': 'pegasus', 'shape': [16]}
Ilość kubitów:  5760
Ilość połączen:  40279
```

1.1 Przykład rozwiązania problemu Max-Cut

1.2 Max-Cut

Problem Max Cut jest jednym z najbardziej znanych problemów optymalizacji kombinatorycznej. Dla danego grafu nieskierowanego $G(V, E)$ ze zbiorem wierzchołków V oraz zbiorem krawędzi E, problem Max Cut polega na podziale zbioru V na dwa podzbiory, tak aby liczba krawędzi łączących te podzbiory (tzw. przeciętych przez cięcie) była jak największa.

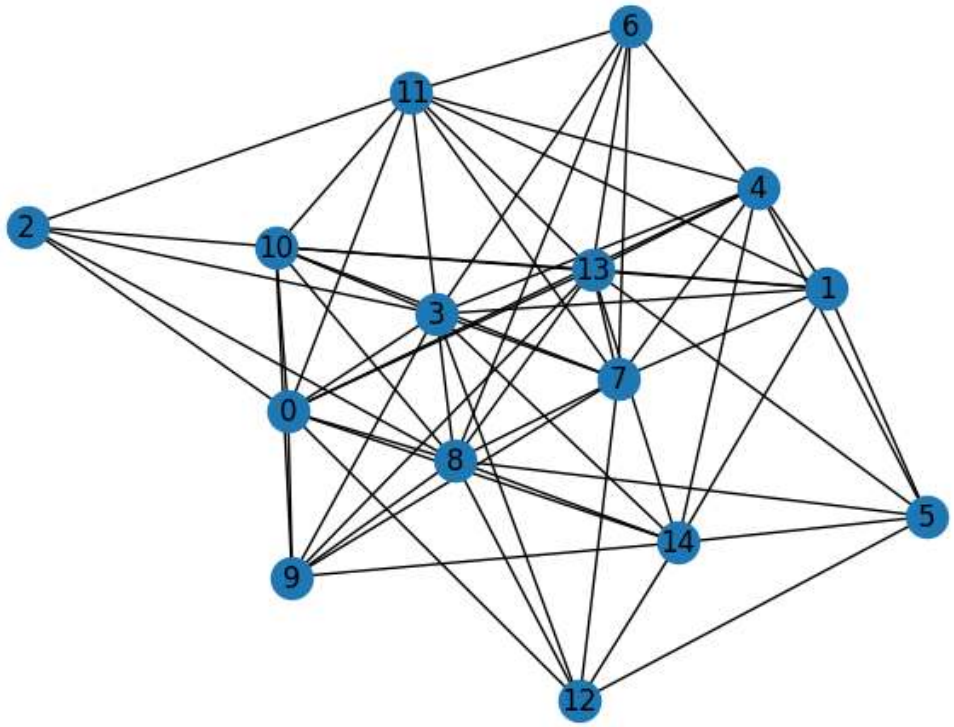
Problem ten można zamodelować za pomocą zmiennych binarnych, gdzie $x_i = 1$, jeśli wierzchołek i

należy do jednego podzbioru oraz $x_i = 0$, gdy należy do drugiego. Krawędź (i, j) jest przecięta przez cięcie (tzn. jej końce znajdują się w różnych podzbiorach), co opisuje wyrażenie $x_i + x_j - 2x_i x_j$.

W związku z tym problem maksymalizacji liczby krawędzi w cięciu można zapisać następująco:

$$\max \sum_{(i,j)\in G} (x_i + x_j - 2x_i x_j) = \min \sum_{(i,j)\in G} (-x_i - x_j + 2x_i x_j)$$

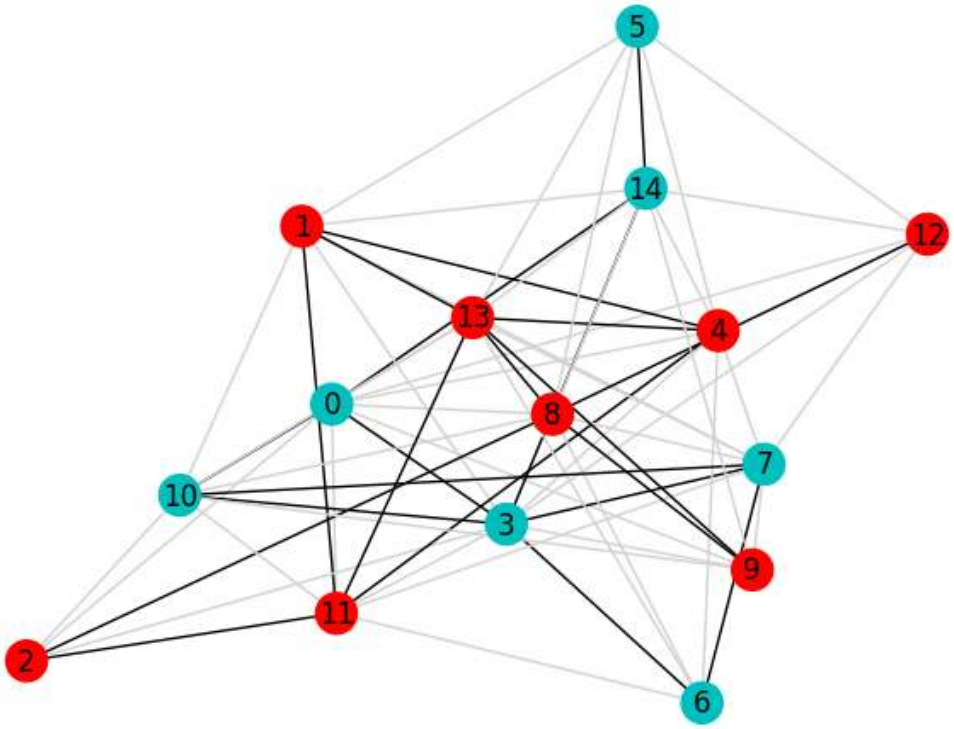
[1]:
```
# Przykład grafu losowego na 15 wierzchołkach
import numpy as np
import networkx as nx

graph = nx.erdos_renyi_graph(n=15, p=0.6, seed=42)
nx.draw(graph, with_labels=True)
```



[9]:
```
# QUBO Zbudujemy QUBO, jako że jest to najnaturalniejsza formulacja
# Warto wspomnieć że zarówno sample_ising() oraz sample_qubo() jako argumenty␣
 ↪przyjmują słowniki
# Jeżeli Twoje dane są w innym formacie, należy użyć BinaryQuadraticModel z␣
 ↪paczki dimod
```

```python
import networkx as nx

from collections import defaultdict
from dwave.system import DWaveSampler, AutoEmbeddingComposite

Q = defaultdict(int)
for i, j in graph.edges:
    Q[(i,i)]+= -1
    Q[(j,j)]+= -1
    Q[(i,j)]+= 2

sampler = DWaveSampler(token=token, solver="Advantage_system4.1")

# automatycznie mapujemy nasz problem na QPU
solver = AutoEmbeddingComposite(sampler)

solution = solver.sample_qubo(Q, num_reads=5, annealing_time=10, label='Example␣
 ↪- Maximum Cut')
print(solution)

best = solution.first.sample

# Interpretacja wyników
S0 = [node for node in graph.nodes if not best[node]]
S1 = [node for node in graph.nodes if best[node]]
cut_edges = [(u, v) for u, v in graph.edges if best[u]!=best[v]]
uncut_edges = [(u, v) for u, v in graph.edges if best[u]==best[v]]

# narysowanie najlepszego znalezionego wyniku
nx.draw(graph, node_color=["r" if i in S0 else "c" for i in list(graph.nodes)],␣
 ↪with_labels=True,
        edge_color=["black" if e in uncut_edges else "lightgray" for e in␣
 ↪list(graph.edges)])
```

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 energy num_oc. chain_b.
0  1  0  0  1  0  1  1  1  0  0  1  0  0  0  1  -42.0       1      0.0
1  1  0  0  1  0  1  0  1  1  0  0  1  0  0  1  -41.0       2      0.0
2  0  1  1  0  1  0  1  0  1  1  0  0  1  1  0  -41.0       1      0.0
3  1  0  0  1  0  1  0  1  1  0  0  1  0  1  1  -40.0       1 0.133333
['BINARY', 4 rows, 5 samples, 15 variables]
```



[13]:
```
# Certyfikacja rozwiązania

from itertools import product
from math import inf
from dimod import BinaryQuadraticModel

qubo = BinaryQuadraticModel(Q, "BINARY")
n = qubo.num_variables

best_energy = inf
best_state = []
for state in product([0, 1], repeat=n):
    x = np.array(state)
    energy = qubo.energy(x)
    if energy < best_energy:
        best_energy = energy
        best_state = state

print("Stan podstawowy: ", best_energy)
print("Znaleziony przez D-Wave: ", solution.first.energy)
```

```
Stan podstawowy:  -42.0
Znaleziony przez D-Wave:  -42.0
```

2 Najlepsze praktyki przy używaniu wyżarzaczy kwantowych

2.1 Balans pomiędzy czasem wyżarzania a ilością odczytów

ako wskazówkę można przyjąć, że opłacalne jest wyrównanie czasu wyżarzania i odczytu. Zwiększenie liczby odczytów może poprawić uzyskiwane rozwiązania, jednak korzyści maleją po przekroczeniu pewnej wartości zależnej od problemu. Tak samo, zwiększenie czasu wyżarzania powyżej pewnej, zaleznej od problemu wartości, również daje malejące korzyści.

Zwiększenie zarówno czasu wyżarzania, jak i liczby odczytów podnosi prawdopodobieństwo sukcesu przy rozwiązywaniu zadanego problemu. Optymalna kombinacja czasu wyżarzania i liczby odczytów dla uzyskania najlepszego rozwiązania w ustalonym budżecie czasowym zależy od konkretnego problemu i wymaga przeprowadzenia eksperymentów.

2.2 Postprocessing

Narzędzia Ocean wykonują pewne minimalne przetwarzanie końcowe. Przykładowp, przerwane łańcuchy (kiedy kubity reprezentujące daną zmienną logiczną mają różne stany) mogą zostać rozstrzygnięte poprzez głosowanie większościowe: Ocean przypisuje wartość zmiennej na podstawie stanu zwróconego przez większość kubitów w łańcuchu.

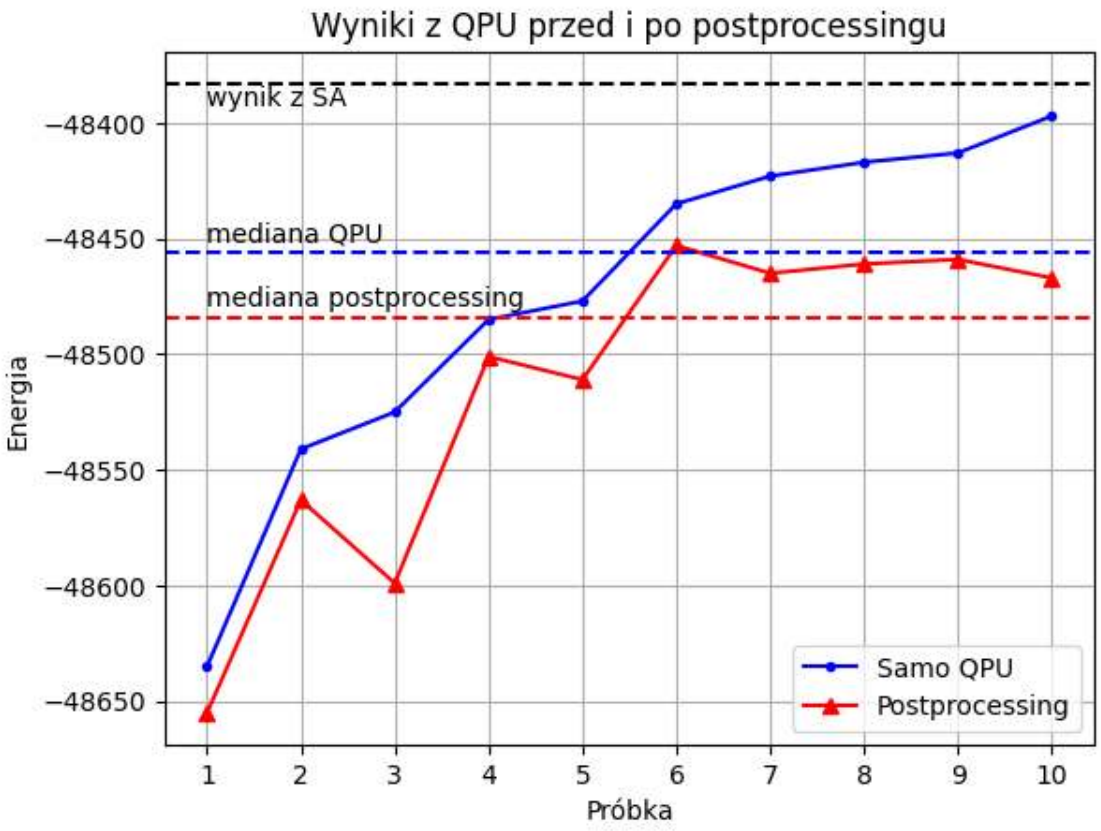
Często możliwe jest poprawienie wyników uzyskanych z wyżarzacha kwantowego niewielkim kosztem przez zastosowanie postprocessingu.

Pakiet dwave-samplers udostępnia implementację solvera bazującego na metodzie gradientu prostego `SteepestDescentSolver`. W tym przykładzie ten klasyczny algorytm jest uruchamiany z próbkami uzyskanymi z QPU, aby znaleźć minima w otoczeniu tych próbek.

[2]:
```
# Przykład
from dwave.system import DWaveSampler
from dwave.samplers import SteepestDescentSolver, SimulatedAnnealingSampler
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.default_rng(seed=42)

# Tworzymy problem wykorzystując wszystkie qubity i celowo dajemy siłę␣
 ↪couplingów z poza naturalnego zasięgu
sampler = DWaveSampler(token=token, solver="Advantage_system4.1")
h = {v: 0.0 for v in sampler.nodelist}
J = {tuple(c): rng.choice(list(range(-5, 6))) for c in sampler.edgelist}

solver_greedy = SteepestDescentSolver()

sampleset_qpu = sampler.sample_ising(h, J,
                                     num_reads=10,
```

```
                                    label='Przyklad - Postprocessing')
# Postprocessing
sampleset_pp = solver_greedy.sample_ising(h, J, initial_states=sampleset_qpu)

mediana_qpu = np.median(sampleset_qpu.record.energy)
mediana_pp = np.median(sampleset_pp.record.energy)

# Certyfikacja
sa_sampler = SimulatedAnnealingSampler()
sampleset_sa = sa_sampler.sample_ising(h, J, num_reads=100)

sa_energy = sampleset_sa.first.energy


print("Mediana energii: ")
print(f"Samo QPU: {mediana_qpu}")
print(f"Postprocessing: {mediana_pp}")



plt.plot(list(range(1, 11)), sampleset_qpu.record.energy, 'b.-')
plt.plot(list(range(1, 11)), sampleset_pp.record.energy, 'r^-')

plt.xticks(list(range(1, 11)))
plt.xlabel("Próbka")
plt.ylabel("Energia")
plt.title("Wyniki z QPU przed i po postprocessingu")
plt.grid()
plt.axhline(mediana_qpu, linestyle="--", color="blue")
plt.axhline(mediana_pp, linestyle="--", color="red")
plt.axhline(sa_energy, linestyle="--", color="black")

plt.text(1, mediana_qpu+5, "mediana QPU", fontsize=10, color='black')
plt.text(1, mediana_pp+5, "mediana postprocessing", fontsize=10, color='black')
plt.text(1, sa_energy-10, "wynik z SA", fontsize=10, color='black')

plt.legend(['Samo QPU', 'Postprocessing'])
plt.show()
```

```
Mediana energii:
Samo QPU: -48456.0
Postprocessing: -48484.0
```

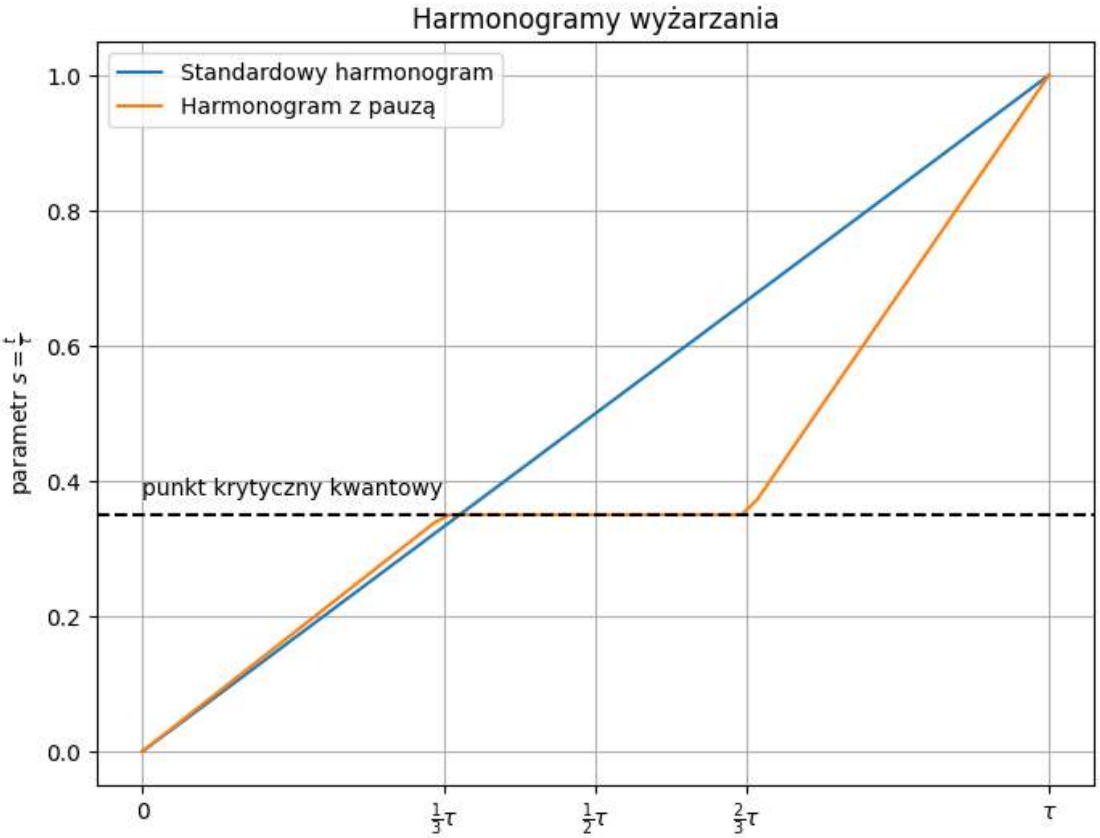


2.3 Harmonogram wyżarzania

Harmonogram wyżarzania w maszynach DWave pozwala nam regulować prędkość z jaką zachodzą zmiany w parametrze s

[1]:
```
# Harmonogram wyżarzania z Pauzą

import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 1, num=60)
y_lin = x
QCP = 0.35
y_pause = np.where(x <= 1/3, 1.05*x,
                   np.where(x <= 2/3, QCP, 1.95*x-0.95))

plt.figure(figsize=(8,6))
plt.plot(x, y_lin, label="Standardowy harmonogram")
plt.plot(x, y_pause, label="Harmonogram z pauzą")
```

```
plt.title("Harmonogramy wyżarzania")
plt.ylabel(r"parametr $s=\frac{t}{\tau}$")
plt.xticks([0, 1/3, 1/2, 2/3, 1], labels=["0", r"$\frac{1}{3} \tau$",␣
 ↪r"$\frac{1}{2} \tau$", r"$\frac{2}{3} \tau$", r"$\tau$"])
plt.grid()
plt.axhline(QCP, linestyle="--", color="black")
plt.text(0, QCP + 0.03, "punkt krytyczny kwantowy")
plt.legend()
plt.show()
```



02_Algorytm_symulowanego_wyżarzania

May 26, 2026

1 Algorytm symulowanego wyżarzania

Symulowane wyżarzanie (*simulated annealing*) to probabilistyczna technika optymalizacyjna inspirowana procesem wyżarzania w metalurgii. Naśladuje fizyczny proces podgrzewania i powolnego schładzania materiału w celu usunięcia defektów. W kontekście optymalizacji „temperatura" jest skalarnym parametrem algorytmu.

1.1 Ogólny zarys

Rozpoczyna się od dowolnie wybranego początkowego rozwiązania i wysokiej temperatury, która jest stopniowo obniżana. Na każdym kroku algorytm losowo modyfikuje bieżące rozwiązanie w celu wygenerowania nowego kandydata. Jeśli nowe rozwiązanie jest lepsze, zostaje zaakceptowane. Jeśli jest gorsze, może również zostać zaakceptowane z pewnym prawdopodobieństwem, które jest proporcjonalne do temperatury. Pozwala to algorytmowi uniknąć lokalnych ekstremów i z czasem przybliżyć rozwiązanie optymalne w skali globalnej.

Tutaj zostanie przedstawiona podstawowa wersja tego algorytmu. Składa się z dwóch pętli. Pierwsza z nich, zewnętrzna, zarządza zmianą temperatury. Druga, wewnętrzna, determinuje ile sąsiadów jest sprawdzanych dla danej temperatury. Dobrą praktyką jest sprawdzenie wszystkich zmiennych w każdym kroku temperatury. Poniżej jest zaprezentowany pseudokod:

Algorithm 1: Simulated Annealing for Ising Model

Data: Initial configuration S_{init}, initial temperature T_0, number of steps K, number of spins N
Result: Approximate solution to the ground state problem
// Initialization
$T \leftarrow T_0$;
$S \leftarrow S_{init}$;
for $k = 0$ **to** K **do**
  $\beta \leftarrow \frac{1}{T}$;
  for $m = 0$ **to** N **do**
    $S' \leftarrow$ neighbor(S);
    $E' \leftarrow E(S')$;
    $\Delta E \leftarrow E(S') - E(S)$;
    if $\Delta E < 0$ **then**
      $S \leftarrow S'$;
    else
      $p \leftarrow P(\Delta E, \beta)$;
      if random$(0, 1) < p$ **then**
        $S \leftarrow S'$;
      end
    end
  end
  $T \leftarrow$ schedule(T) // Update temperature
end
return S // Return the best found solution

Jako funkcji $P(\Delta E, \beta)$ użyjemy tzw. kryterium Metropolisa–Hastingsa:

$$P(\Delta E, \beta) = \begin{cases} 1, & \text{jeżeli } \Delta E \leq 0 \\ e^{-\beta \Delta E}, & \text{w pozostałych przypadkach} \end{cases}$$

Warto tutaj wspomnieć, że dla "wystarczająco powolnej" funkcji schładzającej istnieją matematyczne gwarancje osiągnięcia optymalnego wyniku. Jednakże czas i ilość kroków potrzebna by tą gwarancję osiągnąć często przekracza wyczerpujące przeszukiwanie.

[5]:
```
from setup import setup_paths
setup_paths()

from funkcje_pomocnicze import read_instance, small_grid, P2, small_pegasus,␣
 ↪calculate_energy

instance = small_pegasus

J, h = read_instance(instance.path)
```

```
print("nazwa instancji: ", instance.name)
print("ilość spinów: ", instance.spins)
print("najlepsza energia: ", instance.best_energy)
```

```
nazwa instancji:  P2
ilość spinów:  24
najlepsza energia:  -39.0
```

[24]:
```
# Implementacja algorytmu symulowanego wyżarzania razem z funkcjami pomocniczymi
from setup import setup_paths
setup_paths()

import numpy as np
from math import exp
from funkcje_pomocnicze import calculate_energy


def acceptance_probability(delta_e: float, temp: float):
    if delta_e < 0:
        return 1
    else:
        beta = 1/temp
        probability = exp(-beta * delta_e)

        return probability


def calculate_delta_e(J, h, idx, state):
    s_k = state[idx]
    sum_j = J[idx, :] @ state.T
    return 2*s_k *(h[idx] + sum_j)


def get_temp(J, h):

    # W gorącej temperaturze chcemy by każdy spin mógł się zmienić z␣
 ↪prawdopodobieństwem co najmniej 50%
    # rozwiązujemy:
    #   0.50 = exp(-hot_beta * max_delta_energy)
    # rozwiązaniem jest hot_beta = log(2)/max_delta_energy, czyli T =␣
 ↪max_delta_energy/log(2)
    # gdzie max_delta_energy = 2*max_effective_field, a max_effective_field =␣
 ↪max_i (|h_i| + sum_j |J_ij|)

    h_abs = np.abs(h)
    J_abs = np.abs(J)
    values = [h_abs[i] + sum(J_abs[i, :]) for i in range(len(h))]
```

```
    max_effective_field = max(values)
    hot = (2*max_effective_field) / np.log(2)

    # w zimnej temp zakładamy że jesteśmy w minimum (lub stanie podstawowym) i chcemy ograniczyć
    # prawdopodobieństwo że jakikolwiek spin się zmieni. Dodatkowo dla uproszczenia, zakładamy że tylko
    # spiny o minimalnej luce mogą wejść w stan wzbudzony. Rozwiązujemy:
    #   0.01 ~ #minimal_gaps exp(- cold_beta min_i min_delta_energy_i)
    # gdzie #minimal_gaps to ilość przypadków o minimalnej luce energetycznej
    # min_delta_energy_i = min_i(min_delta_energy_i).
    # rozwiązaniem jest cold_beta = log(#minimal_gaps/0.01) / min_delta_energy czyli:
    # T =  min_delta_energy / log(#minimal_gaps/0.01)


    h_non_zero = h_abs[np.where(h_abs!=0)]
    def J_non_zero(A):
        return A[np.where(A!=0)]

    values_array = [min(h_non_zero[i], np.min(J_non_zero(J_abs[i, :]))) for i in range(len(h_non_zero))]
    min_effective_field = min(values_array)
    cold = (2*min_effective_field) / np.log(1/0.01)

    return hot, cold
```

[25]:
```
# implementacja naiwna
from typing import Optional
from copy import deepcopy

def simulated_annealing_naive(J, h, num_steps: int, temp_range: Optional[tuple] = None, schedule: str = "linear"):
    # inicjalizacja
    n = len(h)
    solution = np.random.choice([-1, 1], size=n)
    energy = calculate_energy(J, h, solution)


    # Jeżeli temperatura nie była podana to jest dobierana automatycznie
    if not temp_range:
        T_0, T_final = get_temp(J, h)
    else:
        T_0, T_final = temp_range


    # Ustawiamy schedule
```

```
    if schedule == "linear":
        schedule = np.linspace(T_0, T_final, num=num_steps, endpoint=True)

    elif schedule == "exponential":
        schedule = np.geomspace(T_0, T_final, num=num_steps, endpoint=True)

    else:
        raise ValueError("Nieprawidłowy schedule")

    # główna pętla algorytmu
    for k in range(num_steps):
        temp = schedule[k]
        for idx in range(n):

            new_solution = deepcopy(solution)
            new_solution[idx] *= -1
            new_energy = calculate_energy(J, h, new_solution)
            delta_e = new_energy - energy

            r = np.random.random()

            if acceptance_probability(delta_e, temp) > r:
                solution = new_solution
                energy = new_energy


    return solution, energy
```

[20]:
```
from setup import setup_paths
setup_paths()

from funkcje_pomocnicze import read_instance, small_grid, P2, small_pegasus,␣
 ↪calculate_energy

instance = P4

J, h = read_instance(instance.path)

state, energy = simulated_annealing_naive(J, h, num_steps=100,␣
 ↪schedule="exponential")

print(energy)
```

```
-363.0
```

Powyższa implementacja będzie działać sprawnie dla małych instancji, ale dla większych czas obliczeń bardzo szybko będzie rosnąć. Warto zauważyć, że w głównej pętli występują dwa bardzo

drogie operacje. Po pierwsze robimy głęboką kopie (`deepcopy`) stanu w każdym kroku. Ponadto dwa razy liczymy energię używając mnożenia macierzy.

By temu zaradzić, zauważamy, że intersuje nas różnica energii pomiędzy rozwiązaniami (ΔE), a nie ich dokładna wartość.

$$\Delta E_k = 2s_k \left(h_k + \sum_j J_{kj} s_j \right)$$

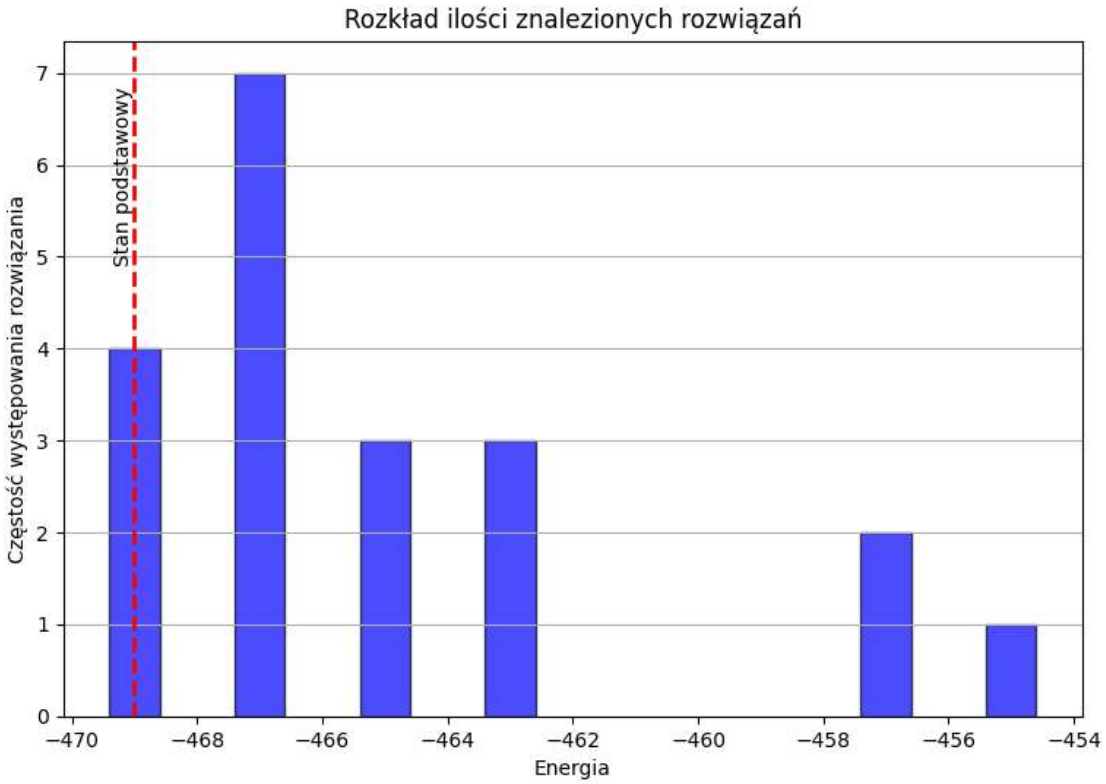
Wyprowadzenie tego wzoru można znaleźć tutaj.

[26]:
```python
def simulated_annealing(J, h, num_steps: int, temp_range: Optional[tuple] =␣
↪None, schedule: str = "linear"):
    # inicjalizacja
    n = len(h)
    solution = np.random.choice([-1, 1], size=n)
    energy = calculate_energy(J, h, solution)


    # Jeżeli temperatura nie była podana to jest dobierana automatycznie
    if not temp_range:
        T_0, T_final = get_temp(J, h)
    else:
        T_0, T_final = temp_range


    # Ustawiamy schedule
    if schedule == "linear":
        schedule = np.linspace(T_0, T_final, num=num_steps, endpoint=True)

    elif schedule == "exponential":
        schedule = np.geomspace(T_0, T_final, num=num_steps, endpoint=True)

    else:
        raise ValueError("Nieprawidłowy schedule")

    for k in range(num_steps):
        temp = schedule[k]
        for s in range(n):
            delta_e = calculate_delta_e(J, h, s, solution)
            r = np.random.random()

            if acceptance_probability(delta_e, temp) > r:
                solution[s] *= -1
                energy += delta_e

    return solution, energy
```

[27]:
```
# Test dla małej instancji

import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance, small_grid, test_pegasus, P8
from tqdm import tqdm

instance = test_pegasus

J, h = read_instance(instance.path)

# Ponieważ jest to probabilistyczny algorytm warto go puścić kilka razy i␣
 ↪wybrać najlepszy wynik
energies = []
for _ in tqdm(range(20), desc="wielokrone symulowane wyżarzanie"):
    state, energy = simulated_annealing(J, h, num_steps=1000,␣
 ↪schedule="exponential")
    energies.append(energy)

print(f"Otrzymana energia: {min(energies)}")
print(f"Najniższa znana energia: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/instance.
 ↪best_energy * 100):.2f}%")

unique_values, counts = np.unique(energies, return_counts=True)


plt.figure(figsize=(9, 6))
plt.bar(unique_values, counts, color='blue', edgecolor='black', alpha=0.7)
#plt.xticks(energies)
plt.axvline(x=test_pegasus.best_energy, color='red', linestyle='--',␣
 ↪linewidth=2)
plt.text(test_pegasus.best_energy, plt.ylim()[1]*0.8, 'Stan podstawowy',␣
 ↪color='black', rotation=90,
         verticalalignment='center', horizontalalignment='right')

plt.xlabel("Energia")
plt.ylabel("Częstość występowania rozwiązania")
plt.title("Rozkład ilości znalezionych rozwiązań")
plt.grid(axis="y")
plt.show()
```

```
wielokrone symulowane wyżarzanie:   0%|          | 0/20 [00:00<?, ?it/s]
wielokrone symulowane wyżarzanie: 100%|      | 20/20 [00:17<00:00,
1.13it/s]
```

```
Otrzymana energia: -469.0
Najniższa znana energia: -469.0
Luka energetyczna: -0.00%
```



1.2 Luka energetyczna

Przy porównywaniu różnych algorytmów (albo różnych implementacji tego samego algorytmu) przydaje się pojęcie **luki energetycznej** (ang. *gap*). Definiuje się ją jako:

$$\text{luka energetyczna} = \frac{E_{\text{best}-E}}{E_{\text{best}}}$$

gdzie E_{best} jest najlepszą znaną energią dla danej instancji. W skrócie, jest to parametr który mówi nam o względnej różnic pomiędzy otrzymanym wynikiem a najlepszym znanym wynikiem.

1.3 Zrównoleglenie

Przez to że algorytm sam w sobie jest stochastyczny nie mamy gwarancji że w trakcie jednego wywołania otrzymamy najlepszy możliwy wynik. Jedym z rozwiązać tego problemu jest zrównoleglenie algorytmu. Oznacza to, że jednocześnie śledzimy wiele "trajektorii" i na końcu obliczeń możemy wybrać najlepszą z nich. Można tego dokonać wywołując algorytm wielokrotnie, ale jest

to nieefektywne. Lepszym rozwiązaniem jest trzymanie wielu stanów ułożonych w macierz. Niech σ_i będzie stanem, wtedy mając M trajektorii:

$$\sigma = \begin{bmatrix} | & | & \cdots & | \\ \sigma_1 & \sigma_2 & \cdots & \sigma_M \\ | & | & \cdots & | \end{bmatrix}$$

Wybór czy trzymamy stany w kolumnach czy w wierszach jest arbitralny. To podejście pozwala nam równolegle liczyć wiele rozwiązań.

Warto zwrócić uwagę, że w tej sytuacji zmiana stanu i prawdopodobieństwo jego zaakceptowania musi być liczone niezależnie dla każdej trajektorii.

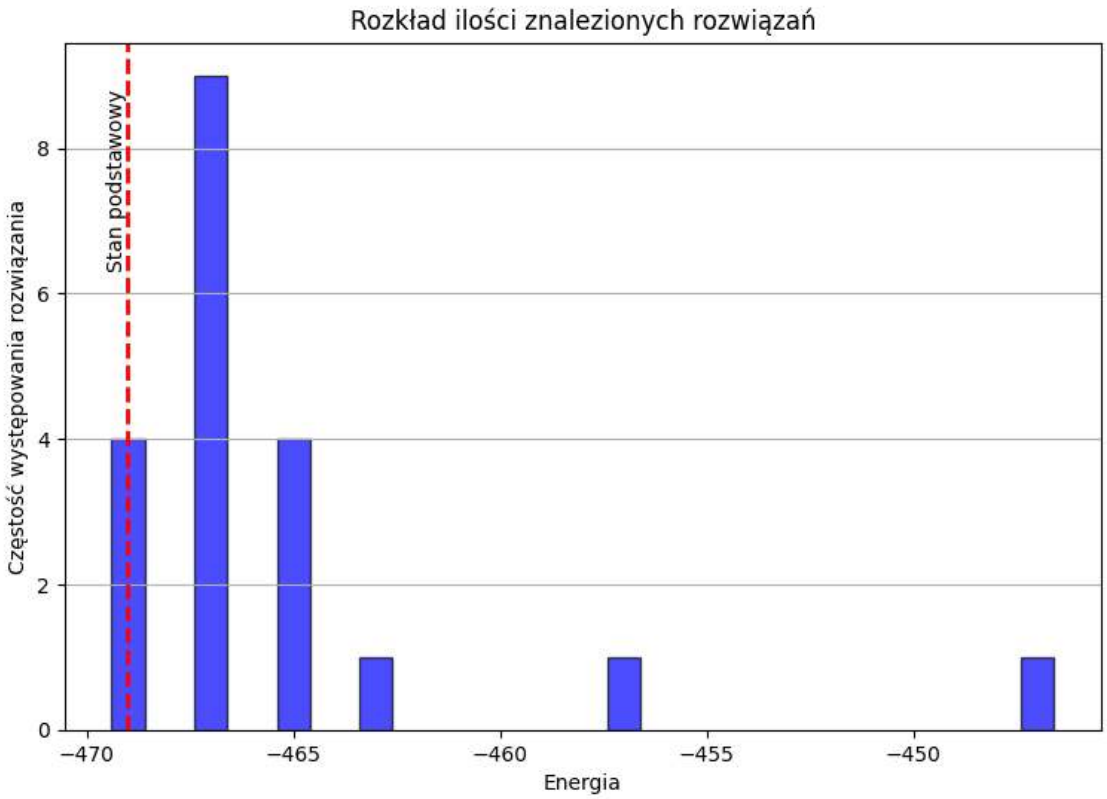
[28]:
```
def calculate_delta_e_matrix(J, h, idx, state, M):
    s_k = state[idx, :]  # <- wiersz
    h_k = np.array([h[idx]] * M)
    sum_j = J[idx, :] @ state
    return 2 * s_k * (h_k + sum_j)


def calculate_energy_matrix(J: np.ndarray, h: np.ndarray, state: np.ndarray,␣
 ↪convention: str = "minus_half"):
    n, _ = J.shape
    if convention == "minus_half":
        A = np.multiply(-1 / 2, J)
        B = np.matmul(A, state) - h.reshape(n, 1)
        C = np.multiply(state, B)
    elif convention == "dwave":
        B = np.matmul(J, state) + h.reshape(n, 1)
        C = np.multiply(state, B)
    return np.sum(C, axis=0)


def acceptance_probability_vect(delta_e, temp, r):
    beta = 1 / temp
    prob = np.ones_like(delta_e)
    mask = delta_e >= 0
    prob[mask] = np.exp(-beta * delta_e[mask])
    accept = prob > r
    return accept


def simulated_annealing_multiple_trajectories(J, h, M: int, num_steps: int,␣
 ↪temp_range: Optional[tuple] = None,
                                  schedule: str = "linear"):
    # inicjalizacja
    n = len(h)
    solution = np.random.choice([-1, 1], size=(n, M))
```

```
    energy = calculate_energy_matrix(J, h, solution)

    # Jeżeli temperatura nie była podana to jest dobierana automatycznie
    if not temp_range:
        T_0, T_final = get_temp(J, h)
    else:
        T_0, T_final = temp_range

    # Ustawiamy schedule
    if schedule == "linear":
        schedule = np.linspace(T_0, T_final, num=num_steps, endpoint=True)

    elif schedule == "exponential":
        schedule = np.geomspace(T_0, T_final, num=num_steps, endpoint=True)

    else:
        raise ValueError("Nieprawidłowy schedule")

    for k in tqdm(range(num_steps), desc= "simulated annealing"):
        temp = schedule[k]
        for idx in range(n):
            delta_e = calculate_delta_e_matrix(J, h, idx, solution, M)
            r = np.random.random(M)
            mask = acceptance_probability_vect(delta_e, temp, r)

            solution[idx, mask] *= -1
            energy[mask] += delta_e[mask]

    return solution, energy
```

[29]:
```
# Test dla małej instancji

import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance, small_grid, test_pegasus, P8
from tqdm import tqdm

instance = test_pegasus

J, h = read_instance(instance.path)

# używamy wersji zrównoleglonej

states, energies = simulated_annealing_multiple_trajectories(J, h, 20,␣
 ↪num_steps=1000, schedule="exponential")

print(f"Otrzymana energia: {min(energies)}")
print(f"Najniższa znana energia: {instance.best_energy}")
```

```
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/instance.
 ↪best_energy * 100):.2f}%")

unique_values, counts = np.unique(energies, return_counts=True)


plt.figure(figsize=(9, 6))
plt.bar(unique_values, counts, color='blue', edgecolor='black', alpha=0.7)
#plt.xticks(energies)
plt.axvline(x=test_pegasus.best_energy, color='red', linestyle='--',␣
 ↪linewidth=2)
plt.text(test_pegasus.best_energy, plt.ylim()[1]*0.8, 'Stan podstawowy',␣
 ↪color='black', rotation=90,
         verticalalignment='center', horizontalalignment='right')

plt.xlabel("Energia")
plt.ylabel("Częstość występowania rozwiązania")
plt.title("Rozkład ilości znalezionych rozwiązań")
plt.grid(axis="y")
plt.show()
```

```
simulated annealing:   0%|          | 0/1000 [00:00<?, ?it/s]
simulated annealing: 100%|     | 1000/1000 [00:05<00:00, 174.65it/s]
Otrzymana energia: -469.0
Najniższa znana energia: -469.0
Luka energetyczna: -0.00%
```



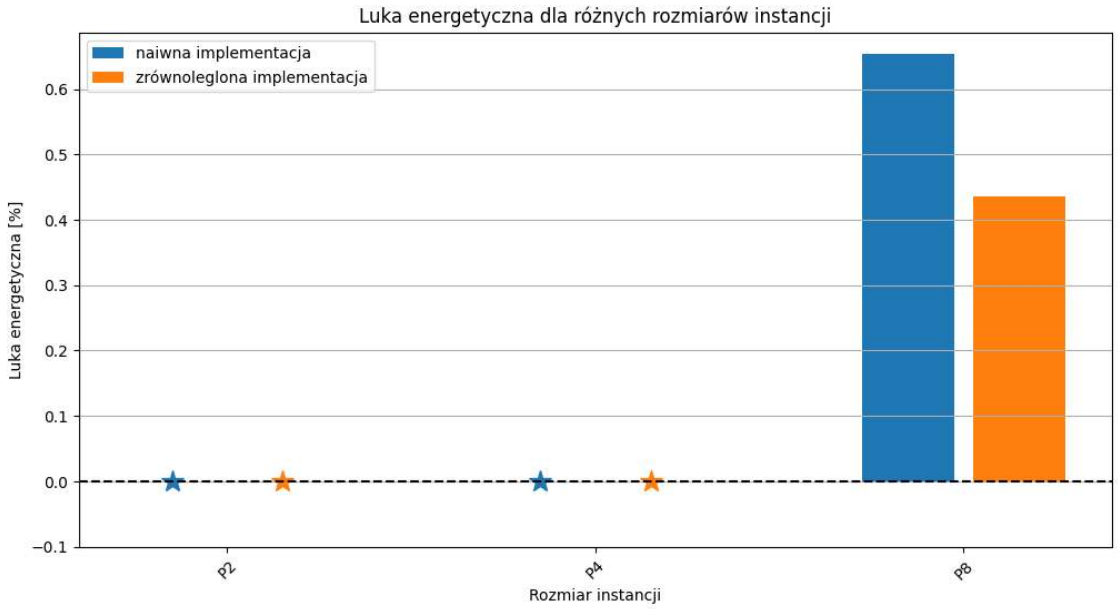
1.4 Porównanie wersji zrównoleglonej z naiwną

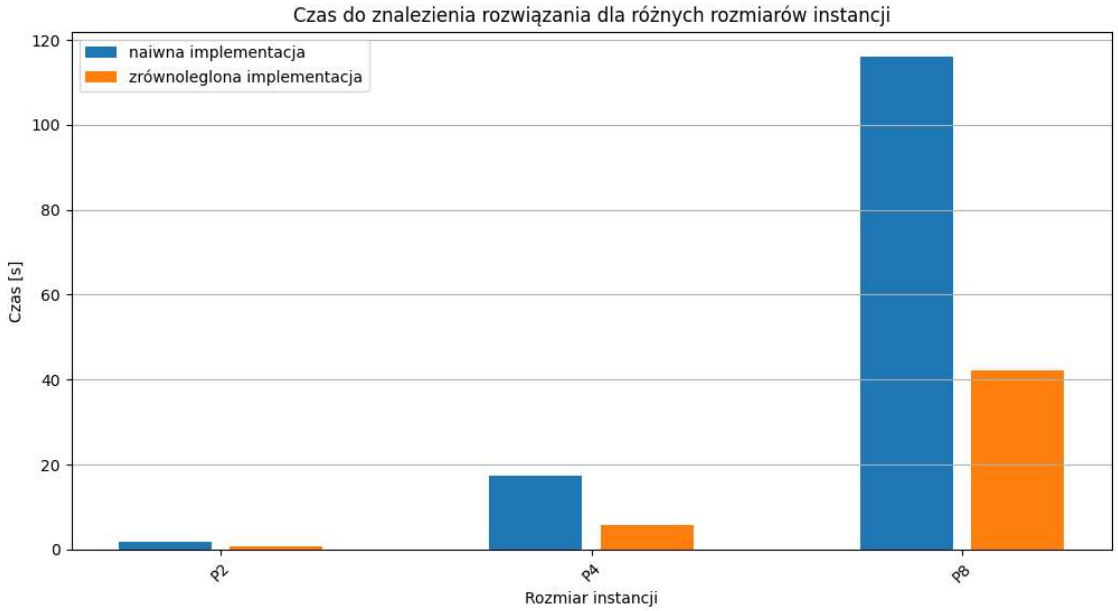
[30]:
```
import time

import matplotlib.pyplot as plt
import numpy as np

from funkcje_pomocnicze import read_instance, P2, P4, P8, P12, P16
from tqdm import tqdm
from math import isclose
from IPython.utils.io import capture_output

num_reads = 20
gaps_naive = []
times_naive = []

gaps = []
times = []


for instance in tqdm([P2, P4, P8], desc="zbieranie_danych"):
```

```
    J1, h1 = read_instance(instance.path)

    begin = time.time()
    with capture_output() as captured:
        states, energies = simulated_annealing_multiple_trajectories(J1, h1,␣
 ↪20, num_steps=1000, schedule="exponential")
    end = time.time()
    sol = min(energies)
    times.append(end-begin)
    gaps.append((instance.best_energy - sol)/instance.best_energy * 100)

    energies = []
    begin = time.time()
    for _ in range(num_reads):
        state, energy = simulated_annealing(J1, h1, num_steps=1000,␣
 ↪schedule="exponential")
        energies.append(energy)
    end = time.time()
    sol = min(energies)
    times_naive.append(end-begin)
    gaps_naive.append((instance.best_energy - sol)/instance.best_energy * 100)

fig, ax = plt.subplots(figsize=(12, 6))

instancje = ["P2", "P4", "P8"]

x = np.arange(len(instancje))
width = 0.25
offset = 0.15

bar1 = ax.bar(x - offset, gaps_naive, width, label="naiwna implementacja")
bar2 = ax.bar(x + offset, gaps, width, label="zrównoleglona implementacja")

barset = [bar1, bar2]
for bars in barset:
    for bar in bars:
        if isclose(bar.get_height(), 0):
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=15, color=bar.
 ↪get_facecolor())

(y_min, y_max) = ax.get_ylim()
ax.set_ylim((-0.1, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
```

```
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji")
ax.axhline(0, color='black', linestyle='--')
#ax.axhline(1, color="red", linestyle="--")
ax.legend()

plt.grid(axis="y")
plt.show()

fig2, ax2 = plt.subplots(figsize=(12, 6))
bar1 = ax2.bar(x - offset, times_naive, width, label="naiwna implementacja")
bars = ax2.bar(x + offset, times, width, label="zrównoleglona implementacja")

ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instancji")
ax2.legend()

plt.grid(axis="y")
plt.show()
```

```
zbieranie_danych: 100%|      | 3/3 [03:03<00:00, 61.29s/it]
```





1.4.1 Heurystyczność

Trzeba tutaj podkreślić, że w wypadku używania algorytmów heurystycznych (np. symulowane wyżarzanie), ich skuteczność i wydajność w dużej mierze zależą od właściwego doboru hiperparametrów. Wybór tych parametrów jest procesem trudnym i często kosztowym obliczeniowo. Czasami prawdziwe trudności obliczeniowe są w ukryte właśnie za decyzjami związanymi z hiperparametryzacją.

Zagadnienia te będą omawiane szerzej w kontekście bardziej złożonych algorytmów, w których pojawiają się dodatkowe aspekty, takie jak możliwość równoległego przetwarzania danych czy wykorzystanie mocy obliczeniowej GPU, co pozwala na znaczne przyspieszenie obliczeń, ale jednocześnie zwiększa złożoność całego procesu optymalizacji.

2 Bibliografia

- Eglese, R.W. (1990). Simulated annealing: A tool for operational research. *European Journal of Operational Research*, Volume **46**, Issue **3**, 1990, DOI: 10.1016/0377-2217(90)90001-R
- Henderson, D., Jacobson, S.H., Johnson, A.W. (2003). The Theory and Practice of Simulated Annealing. In: Glover, F., Kochenberger, G.A. (eds) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol 57. Springer, Boston, MA. DOI: 10.1007/0-306-48056-5_10
- Vodeb, J., Eržen, V., Hrga, T., Povh, J. (2024). Accuracy and Performance Evaluation of Quantum, Classical and Hybrid Solvers for the Max-Cut Problem. *ArXiv preprint*, DOI: 10.48550/arXiv.2412.07460

3 Dodatkowe materiały

- Simulated annealing: From basics to applications

Algorytm symulowanej bifurkacji

Jest to heurystyczny algorytm zainspirowany fizyką, zaproponowany przez Goto *et al.,* w 2019 roku. Jak pokazują ostatnie badania, jest to SOTA algorytm do rozwiązywania problemu Isinga (Pawlowski 2025).

Intuicyjnie, możemy o nim myśleć jako zakodowaniu naszego problemu uoptymalizacjyjnego jako pewien układ dynamiczny opisany równaniami różniczkowymi. Te równania wywodzą się z równiań Hamiltona z mechaniki klasycznej. Intuicuyjnie opisują ewolucję położenia x i pędu y pewnej grupy cząsteczek.

$$ \dot{y}_i = -[a_0 - a(t)]x_i + c_0f_i $$ $$ \dot{x}_i = a_0 y_i $$

gdzie kropka opisuje pochodną po czasie, a_0 jest stałą, $a(t)$ jest parametrem bifurkacji rosnącym w czasie t od 0 do a_0, a c_0 jest parametrem normalizacyjnym zależym od sił f_i. Siły f_i są de facto gradientami energi. Co jest ważne, statnie stacjonarnym, po zbinaryzowaniu te równania opisują lokalne minimum modelu Isinga.

Rozwiązując te równania różniczkowe numerycznie (np. metodą symplektyczną Eulera) otrzymujemy rozwiązanie oryginalnego problemu. Dla naszych zastosowań możemy go traktować jako czarną skrzynkę.

Zostaną Pokazane dwie wersje tego algorytmu które nieznacznie się od siebie różnią, balistyczna symulowana bifurkacja i dyskretna symulowana bifurkacja. Te algorytmy w bardzo naturalny sposób się zrównoleglają.

Uwaga na marginesie

W dodatkowym pliku jest wprowadzenie do rozwiązywania równań różniczkowych numerycznie metodą Eulera i symplektyczną metodą Eulera.

Balistyczna Symulowana Bifurkacja

W każdym kroku t_k "pozycja" $x(t_k)$ i "pęd" $y(t_k)$ są aktualizowane za pomocą następujących wzorów:

$$ y_i(t_{k+1}) = y_i(t_k) + \{-[a_0 - a(t_k)]x_i(t_k) + c_0 \sum_{j=1}^N J_{i,j}x_j(t_k) \}\Delta_t $$ $$ x_i(t_{k+1}) = x_i(t_k) + a_0 y_i(t_{k+1})\Delta_t $$

Oznaczenia są identyczne jak wcześniej. Przez Δ_t oznaczamy długość kroku czasowego. Ponadto przyjmujemy tzw. "nieelastyczną scianę" dla $\vert x \vert = 1$. Oznacza to, że jeżeli w dowolnym momęcie algorytmu $\vert x \vert > 1$ to obcinamy

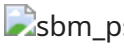
x do odpowiedniej wartości (-1 lub 1) i ustawiamy $y = 0$. Można o tym myśleć jak o cząsteczce uderzającej w ścianie i się do niej przyklejającą. Matematycznie możemy tą operacje wyrazić funkcją $\text{Wall}(x, y)$.

$$ \text{Wall}(x, y) = \left\{ \begin{array}{ll} (\text{sign}(x), 0), & \text{jeżeli } \vert x \vert > 1 \\ (x, y) & \text{w pozostałych przypadkach}\\ \end{array} \right. $$

Na końcu obliczeń należy zbinaryzować "pozycję" by otrzymać stan Isinga w rozwiązaniu. Użyjemy do tego funkcji znaku $\text{sign}(x)$. Dla przypomnienia:

$$ \text{sign}(x) = \left\{ \begin{array}{ll} 1, & \text{jeżeli } x > 0 \\ 0, & \text{jeżeli } x = 0\\ -1, & \text{jeżeli } x < 0\\ \end{array} \right. $$

Pseudokod



In [1]:
```
from setup import setup_paths
setup_paths()

import numpy as np

from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze import calculate_energy

def wall(x: np.ndarray, y: np.ndarray):
    mask = np.abs(x) > 1
    x[mask] = np.sign(x[mask])
    y[mask] = 0
    return x, y


def balistic_simulated_bifurcation_single(J, h, num_steps, time_step,
                                          a_0: Optional[float] = None, c_0_scaling:
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = np.sqrt(np.sum(np.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling


    a = np.linspace(0, a_0, num=num_steps)

    x = np.zeros(N)
    y = np.random.uniform(-0.1, 0.1, N)
```

```
    for t in tqdm(range(num_steps), desc="Symulowana Bifurkacja"):
        y += (-1 * (a_0 - a[t]) * x + c_0 * (J @ x + h )) * time_step  # y(t
        x += a_0 * y * time_step # x(t + 1); y(t+1)

        x, y = wall(x, y)

    x = np.sign(x)
    return x, calculate_energy(J, h, x)
```

In [2]:
```
# test dla małej instancji
import numpy as np
import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance, test_pegasus

J, h = read_instance(test_pegasus.path, convention="minus_half")

state, energy = balistic_simulated_bifurcation_single(J, h, num_steps=200, t

print(f"Otrzymana energia: {energy}")
print(f"Stan podstawowy: {test_pegasus.best_energy}")
```

```
Symulowana Bifurkacja: 100%|██████████| 200/200 [00:00<00:00, 51536.57it/s]
Otrzymana energia: -467.0
Stan podstawowy: -469.0
```

Zrównoleglenie

Przez to że algorytm sam w sobie jest stochastyczny nie mamy gwarancji że w trakcie jednego wywołania otrzymamy najlepszy możliwy wynik. Jedym z rozwiązać tego problemu jest zrównoleglenie algorytmu. Oznacza to, że jednocześnie śledzimy wiele "trajektorii" i na końcu obliczeń możemy wybrać najlepszą z nich. Można tego dokonać wywołując algorytm wielokrotnie, ale jest to nieefektywne. Lepszym rozwiązaniem jest trzymanie wielu stanów ułożonych w macierz. Niech $\bm{x}$ będzie wektorem zmiennych x_i, wtedy mając M trajektorii:

$$ \sigma = \begin{bmatrix} \vert & \vert & \cdots & \vert \\ \bm{x}_1 & \bm{x}_2 & \cdots & \bm{x}_M \\ \vert & \vert & \cdots & \vert \end{bmatrix} $$

Wybór czy trzymamy wektory w kolumnach czy w wierszach jest arbitralny. To podejście pozwala nam równolegle liczyć wiele rozwiązań i jest pełną formą tego algorytmu. Wtedy

W podobny sposób trzymamy y.. Będzie to wymagało pewnych technicznych zmian w kodzie, ale główna idea pozostaje niezmieniona.

In [3]:
```
# Implementacja Balistycznej Symulowanej Bifurkacji dla wielu trajektorii
from setup import setup_paths
setup_paths()
```

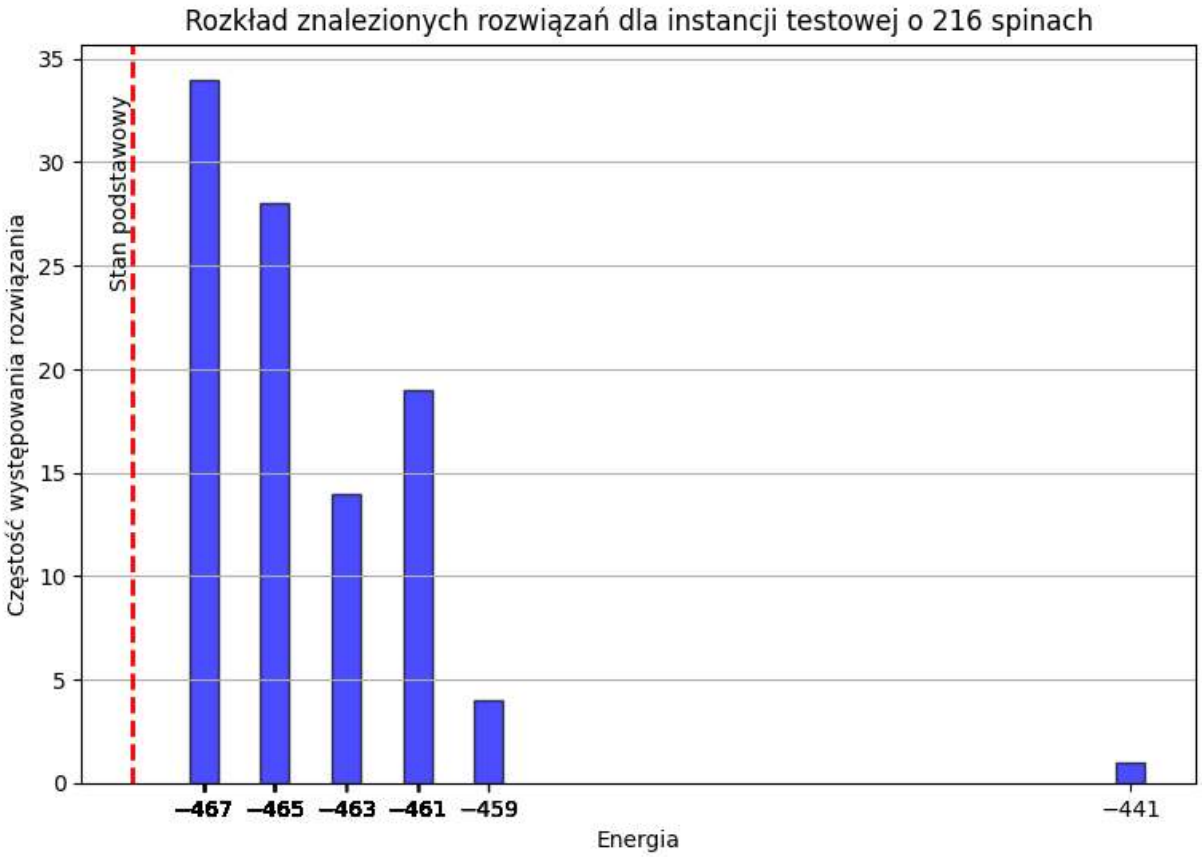
```
import numpy as np

from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze import calculate_energy_matrix

def wall(x: np.ndarray, y: np.ndarray):
    mask = np.abs(x) > 1
    x[mask] = np.sign(x[mask])
    y[mask] = 0
    return x, y


def balistic_simulated_bifurcation(J, h, num_steps, time_step, num_trajector
                                   a_0: Optional[float] = None, c_0_scaling:
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = np.sqrt(np.sum(np.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling


    a = np.linspace(0, a_0, num=num_steps)

    x = np.zeros((N, num_trajectories))
    y = np.random.uniform(-0.1, 0.1, (N, num_trajectories))


    for t in tqdm(range(num_steps), desc="Symulowana Bifurkacja"):
        y += (-1 * (a_0 - a[t]) * x + c_0 * (J @ x + h.reshape((N, 1)))) * t
        x += a_0 * y * time_step # x(t + 1); y(t+1)

        x, y = wall(x, y)

    x = np.sign(x)
    return x, calculate_energy_matrix(J, h, x)
```

In [4]:

```
# Test na małym przykładzie z 216 spinami
# Balistyczna symulowana Bifurkacja

import numpy as np
import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance, test_pegasus, Grid100

instance = test_pegasus

J, h = read_instance(instance.path, convention="minus_half")

states, energies = balistic_simulated_bifurcation(J, h, num_steps=200, time_
```

```
print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/(2 * ins

unique_values, counts = np.unique(energies, return_counts=True)


plt.figure(figsize=(9, 6))
plt.bar(unique_values, counts, color='blue', edgecolor='black', alpha=0.7)
plt.xticks(energies)
plt.axvline(x=instance.best_energy, color='red', linestyle='--', linewidth=2
plt.text(instance.best_energy, plt.ylim()[1]*0.8, 'Stan podstawowy', color='
         verticalalignment='center', horizontalalignment='right')

plt.xlabel("Energia")
plt.ylabel("Częstość występowania rozwiązania")
plt.title("Rozkład znalezionych rozwiązań dla instancji testowej o 216 spina
plt.grid(axis="y")
plt.show()
```

```
Symulowana Bifurkacja:   0%|          | 0/200 [00:00<?, ?it/s]
Symulowana Bifurkacja: 100%|██████████| 200/200 [00:00<00:00, 3496.08it/s]
Otrzymana energia: -467.0
Stan podstawowy: -469.0
Luka energetyczna: 0.21%
```



Dyskretna Symulowana Bifurkacja

Jest podobna do balistycznej, jedyna różnica jest taka, że podczas update parametru y część $J_{i,j}x_j(t_k)$ zastępujemy na $J_{i,j}\text{sign}(x_j(t_k))$, gdzie sign jest funkcją znaku.

Wtedy równania które opisują ewolucję parametrów x i y wyglądają następująco:

$$ y_i(t_{k+1}) = y_i(t_k) + \{-[a_0 - a(t_k)]x_i(t_k) + c_0 \sum_{j=1}^N J_{i,j}\text{sign}(x_j(t_k)) \}\Delta_t $$ $$ x_i(t_{k+1}) = x_i(t_k) + a_0 y_i(t_{k+1})\Delta_t $$

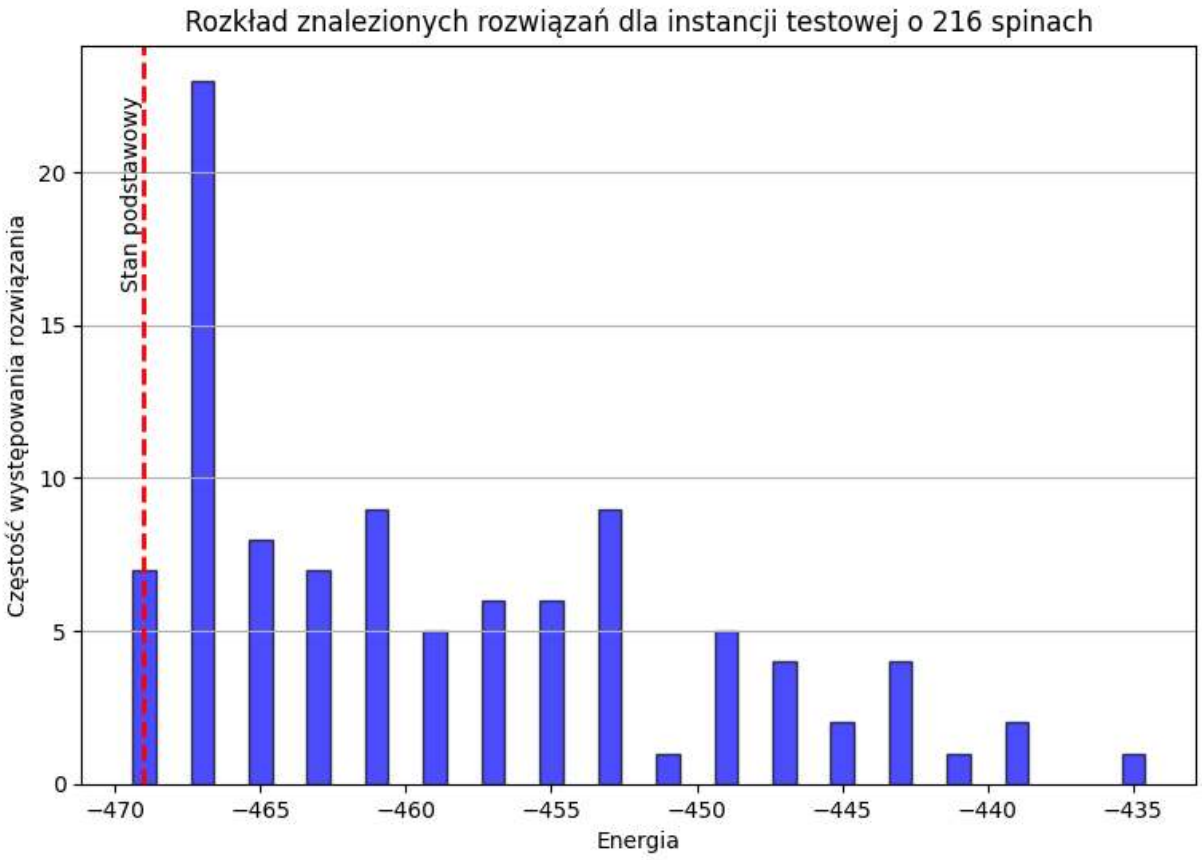
Reszta algorytmu pozostaje bez zmian.

In [5]:
```
# Implementacja dyskretnej Symulowanej Bifurkacji
from setup import setup_paths
setup_paths()


import numpy as np

from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze import calculate_energy_matrix


def wall(x: np.ndarray, y: np.ndarray):
    mask = np.abs(x) > 1
    x[mask] = np.sign(x[mask])
    y[mask] = 0
    return x, y


def discrete_simulated_bifurcation(J, h, num_steps, time_step, num_trajector
                                   a_0: Optional[float] = None, c_0_scaling:
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = np.sqrt(np.sum(np.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling

    a = np.linspace(0, a_0, num=num_steps)

    x = np.zeros((N, num_trajectories))
    y = np.random.uniform(-0.1, 0.1, (N, num_trajectories))

    for t in tqdm(range(num_steps), desc="Symulowana Bifurkacja"):
        y += (-1 * (a_0 - a[t]) * x + c_0 * (J @ np.sign(x) + h.reshape((N,
        x += a_0 * y * time_step # x(t + 1); y(t+1)

        x, y = wall(x, y)
```

```
    x = np.sign(x)
    return x, calculate_energy_matrix(J, h, x)
```

In [6]:
```
# Test na małym przykładzie z 216 spinami
# Balistyczna symulowana Bifurkacja

import numpy as np
import matplotlib.pyplot as plt
from funkcje_pomocnicze import read_instance, test_pegasus

J, h = read_instance(test_pegasus.path, convention="minus_half")

states, energies = discrete_simulated_bifurcation(J, h, num_steps=200, time_

print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {test_pegasus.best_energy}")

unique_values, counts = np.unique(energies, return_counts=True)


plt.figure(figsize=(9, 6))
plt.bar(unique_values, counts, color='blue', edgecolor='black', alpha=0.7)
#plt.xticks(energies)
plt.axvline(x=test_pegasus.best_energy, color='red', linestyle='--', linewic
plt.text(test_pegasus.best_energy, plt.ylim()[1]*0.8, 'Stan podstawowy', col
         verticalalignment='center', horizontalalignment='right')

plt.xlabel("Energia")
plt.ylabel("Częstość występowania rozwiązania")
plt.title("Rozkład znalezionych rozwiązań dla instancji testowej o 216 spina
plt.grid(axis="y")
plt.show()
```

```
Symulowana Bifurkacja:   0%|          | 0/200 [00:00<?, ?it/s]
Symulowana Bifurkacja: 100%|██████████| 200/200 [00:00<00:00, 1999.01it/s]
Otrzymana energia: -469.0
Stan podstawowy: -469.0
```

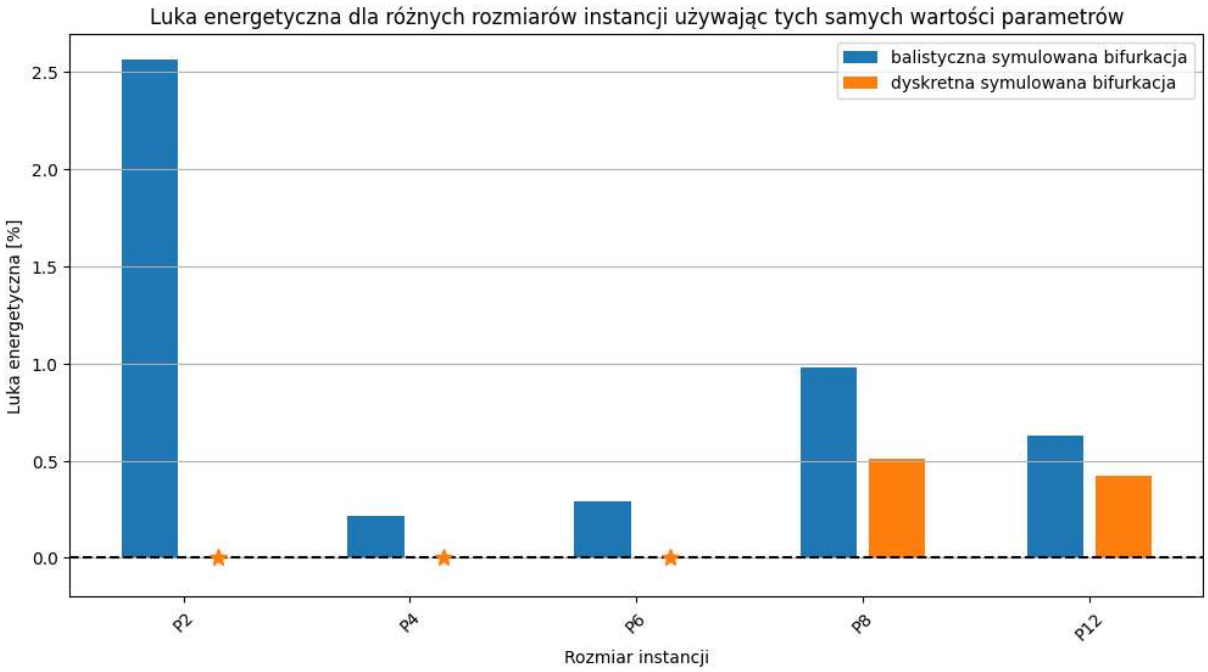


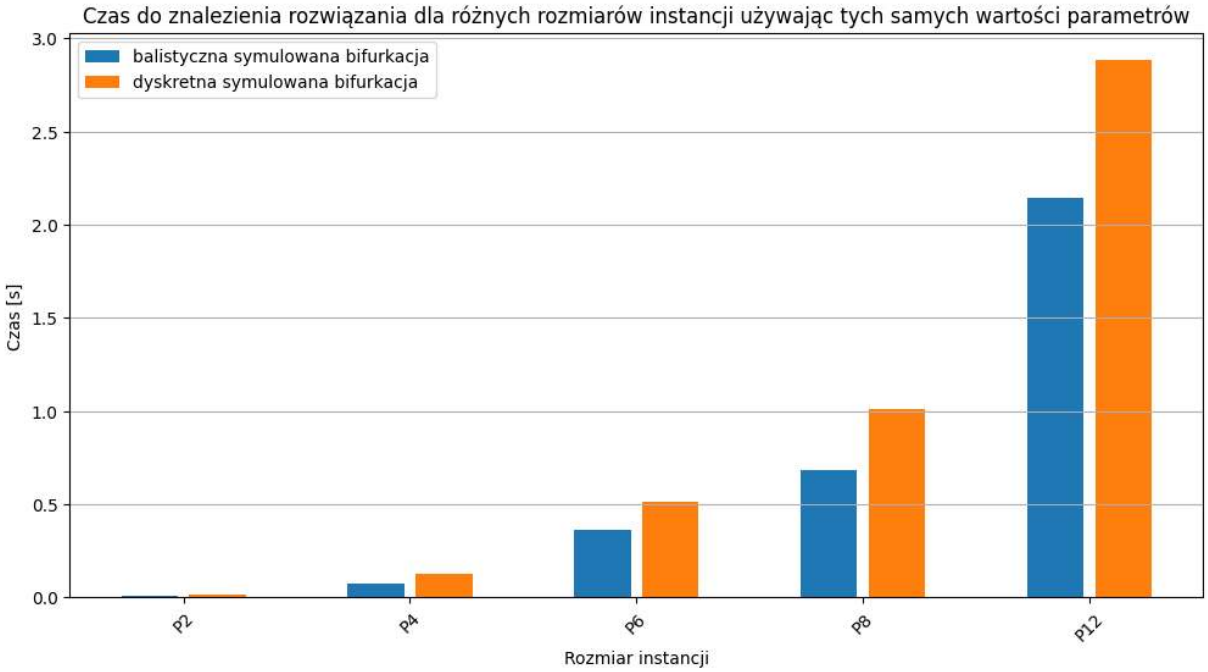
In [7]:
```python
#  Porównanie jakości rozwiązań i czasu obliczeń dla obydwu wersji

import time
import matplotlib.pylab as plt
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8, P12
from IPython.utils.io import capture_output

num_traj = 2**7
num_steps = 200
time_step = 0.75

instances = [P2, P4, P6, P8, P12]

gaps = {}
times = {}

for instance in tqdm(instances, desc="Zbieranie danych"):

    J, h = read_instance(instance.path, convention="minus_half")

    with capture_output() as captured:
        begin = time.time()
        _, energies1 = balistic_simulated_bifurcation(J, h, num_steps=num_st
                                                      time_step=time_step, num
        end = time.time()
        elapsed_balistic = end - begin

        begin = time.time()
        _, energies2 = discrete_simulated_bifurcation(J, h, num_steps=num_st
```

```
                                        time_step=time_step, num_trajectorie
        end = time.time()
        elapsed_discrete = end - begin

    gap_balistic = (instance.best_energy - min(energies1)) / (2*instance.bes
    gap_discrete = (instance.best_energy - min(energies2)) /(2*instance.best

    gaps[(instance.name, "balistyczna")] = gap_balistic
    gaps[(instance.name, "dyskretna")] = gap_discrete

    times[(instance.name, "balistyczna")] = elapsed_balistic
    times[(instance.name, "dyskretna")] = elapsed_discrete

instancje = [instance.name for instance in instances]

x = np.arange(len(instancje))
width = 0.25
offset = 0.15

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-1, 1]
for idx, s in enumerate(["balistyczna", "dyskretna"]):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"{s} symulowana
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
if y_max <1.1:
    y_max = 1.1
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji używając tyc
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()

fig2, ax2 = plt.subplots(figsize=(12, 6))
for idx, s in enumerate(["balistyczna", "dyskretna"]):
    a = [times[(i, s)] for i in instancje]
    bar = ax2.bar(x + offset * param[idx], a, width, label=rf"{s} symulowana

ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
```

```
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instanc
ax2.legend()

plt.grid(axis="y")
plt.show()
```

```
Zbieranie danych: 100%|██████████| 5/5 [00:09<00:00,  1.90s/it]
```





Wpływ parametrów na zachowanie algorytmu

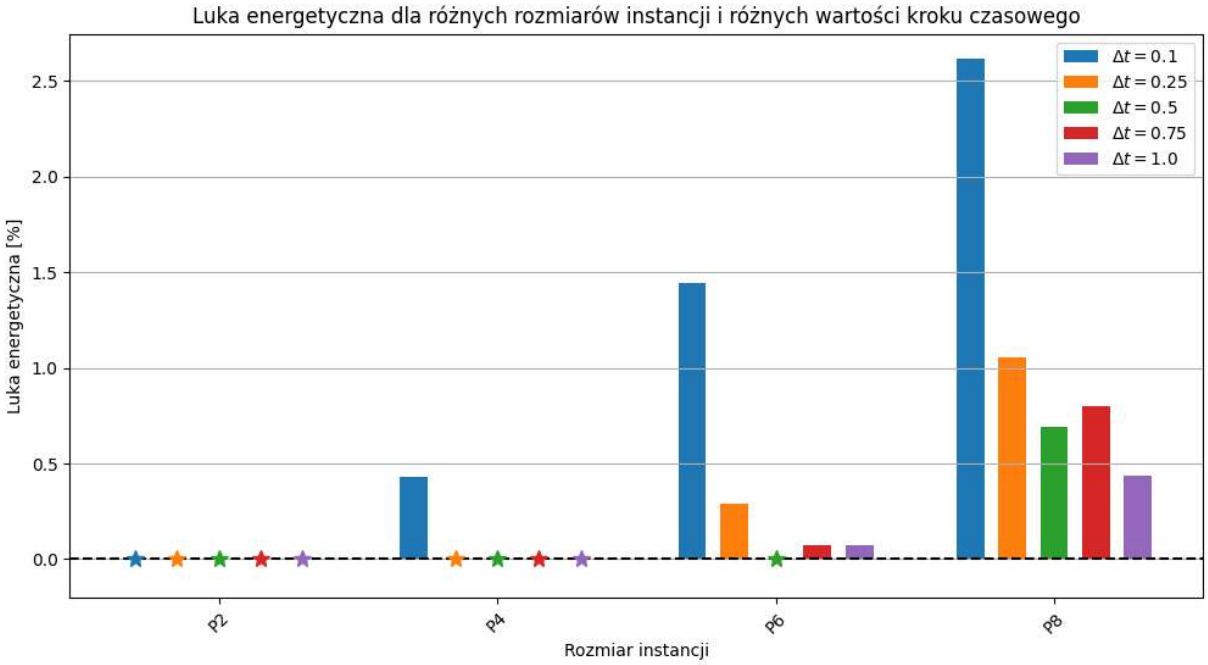
Skupimy się na dyskretnej symulowanej bifurkacji.

In [8]:

```
# krok czasowy delta t

import time
import matplotlib.pylab as plt
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8
from IPython.utils.io import capture_output

num_traj = 2**7
num_steps = 200

time_steps = [0.1, 0.25, 0.5, 0.75, 1.0]
instances = [P2, P4, P6, P8]

gaps = {}
times = {}

with tqdm(total=len(instances)*len(time_steps), desc="zbieranie danych") as
    for t in time_steps:
        for instance in instances:
            J, h = read_instance(instance.path, convention="minus_half")
            with capture_output() as captured:
                begin = time.time()
                _, energies = discrete_simulated_bifurcation(J, h, num_steps
                                                    time_step=t, num_trajectorie
                end = time.time()
                elapsed_discrete = end - begin

            gap = (instance.best_energy - min(energies)) / (2*instance.best_
            if gap < 0:
                print(instance.name, min(energies))
            gaps[(instance.name, t)] = gap
            times[(instance.name, t)] = elapsed_discrete
            pbar.update(1)

instancje = [instance.name for instance in instances]

x = np.arange(len(instancje))
width = 0.1
offset = 0.15

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-2, -1, 0, 1, 2]
for idx, s in enumerate(time_steps):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"$\Delta t ={s}
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
if y_max <1.1:
    y_max = 1.1
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
```

```
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji i różnych wa
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()
```

```
zbieranie danych: 100%|██████████| 20/20 [00:10<00:00,  1.92it/s]
```

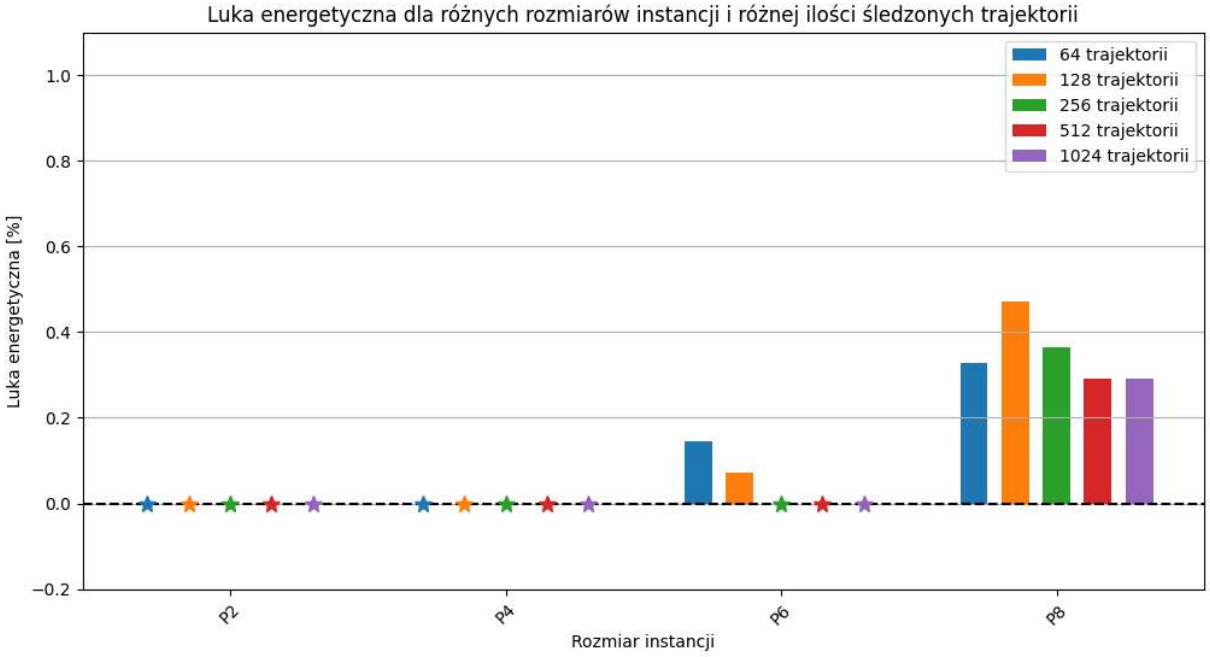


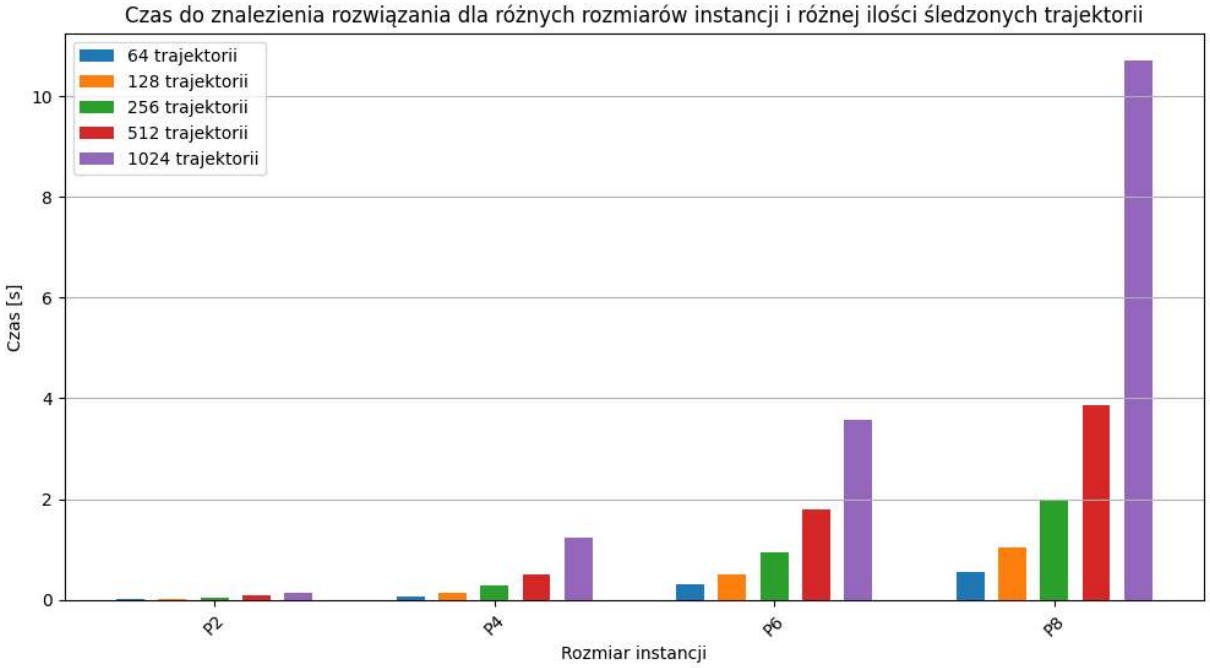
In [9]:
```
# Ilość trajektorii
import matplotlib.pylab as plt
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8
from IPython.utils.io import capture_output
from tqdm import tqdm

time_step = 0.75
num_steps = 200


instances = [P2, P4, P6, P8]
trajectories = [2**6, 2**7, 2**8, 2**9, 2**10]

gaps = {}
times = {}

with tqdm(total=len(instances)*len(trajectories), desc="zbieranie danych") a
    for t in trajectories:
        for instance in instances:
```

```
            J, h = read_instance(instance.path, convention="minus_half")
            with capture_output() as captured:
                begin = time.time()
                _, energies = discrete_simulated_bifurcation(J, h, num_steps
                                                    time_step=time_step, num_tra
                end = time.time()
                elapsed_discrete = end - begin

            gap = (instance.best_energy - min(energies)) / (2*instance.best_
            if gap < 0:
                print(instance.name, min(energies))
            gaps[(instance.name, t)] = gap
            times[(instance.name, t)] = elapsed_discrete
            pbar.update(1)

instancje = [instance.name for instance in instances]

x = np.arange(len(instancje))
width = 0.1
offset = 0.15

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-2, -1, 0, 1, 2]
for idx, s in enumerate(trajectories):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"{s} trajektorii
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
if y_max <1.1:
    y_max = 1.1
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji i różnej ilo
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()

fig2, ax2 = plt.subplots(figsize=(12, 6))
for idx, s in enumerate(trajectories):
    a = [times[(i, s)] for i in instancje]
    bar = ax2.bar(x + offset * param[idx], a, width, label=rf"{s} trajektori
```

```
ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instanc
ax2.legend()

plt.grid(axis="y")
plt.show()
```

```
zbieranie danych: 100%|██████████| 20/20 [00:29<00:00,  1.48s/it]
```



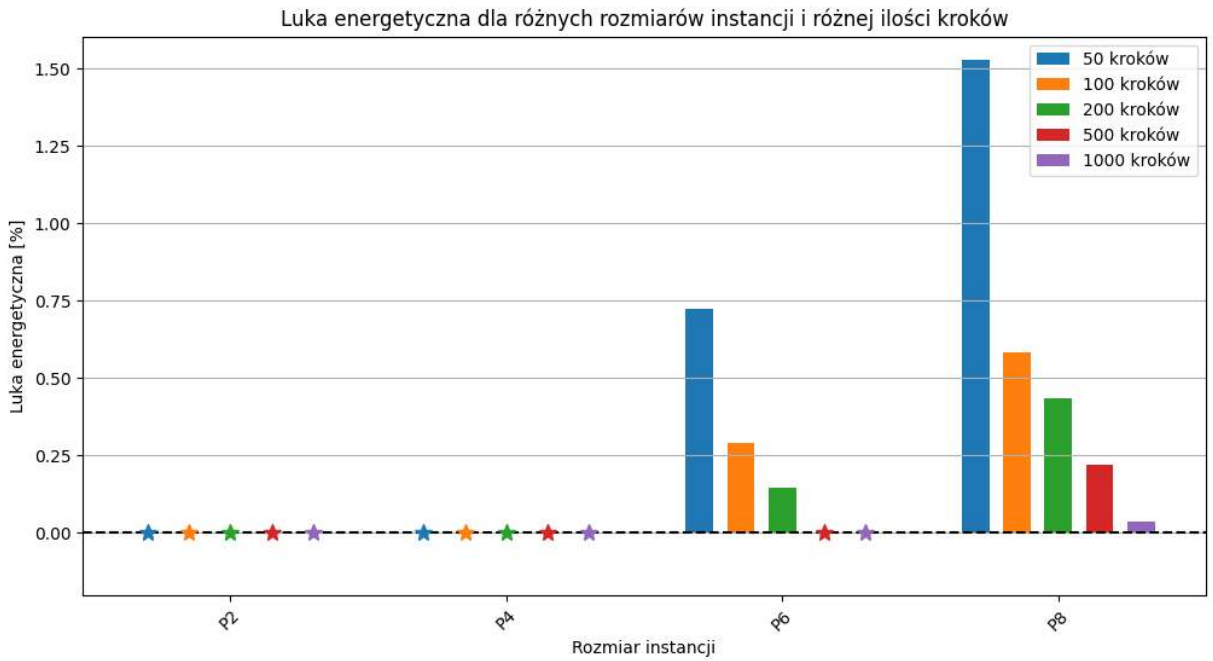


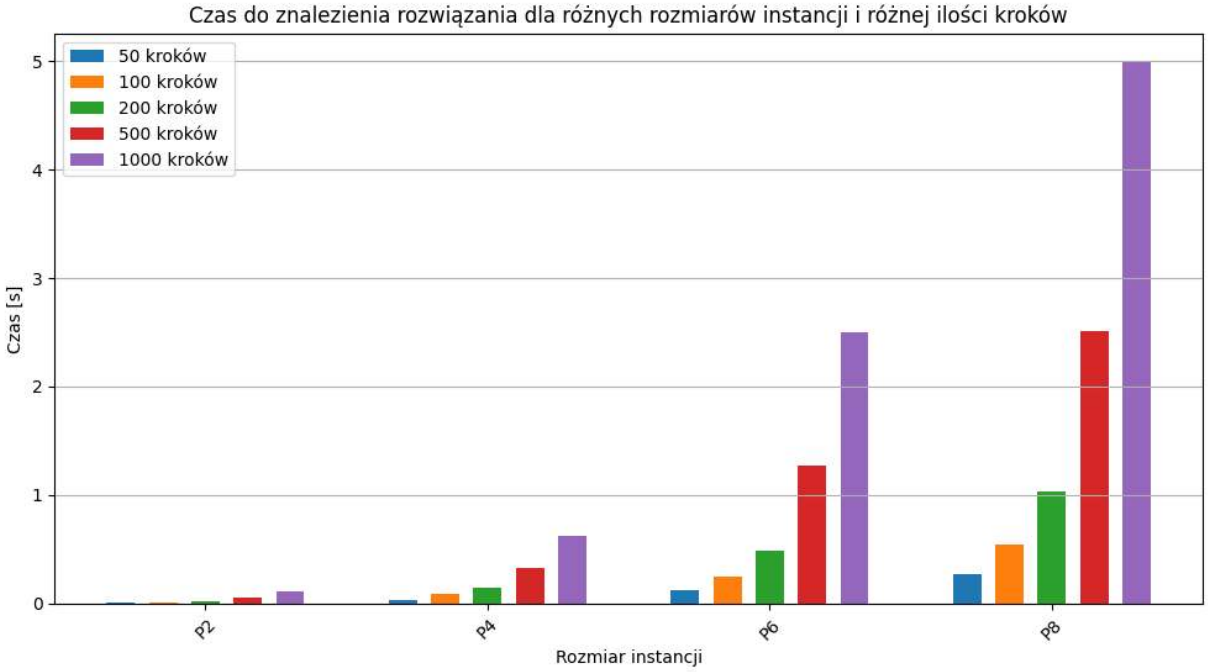
In [10]:
```
# Ilość kroków

import matplotlib.pylab as plt
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8
```

```
from IPython.utils.io import capture_output
from tqdm import tqdm

time_step = 0.75
num_trajectories = 2**7


instances = [P2, P4, P6, P8]
steps = [50, 100, 200, 500, 1000]

gaps = {}
times = {}

with tqdm(total=len(instances)*len(steps), desc="zbieranie danych") as pbar:
    for t in steps:
        for instance in instances:
            J, h = read_instance(instance.path, convention="minus_half")
            with capture_output() as captured:
                begin = time.time()
                _, energies = discrete_simulated_bifurcation(J, h, num_steps
                                                time_step=time_step, num_tra
                end = time.time()
                elapsed_discrete = end - begin

            gap = (instance.best_energy - min(energies)) / (2*instance.best_
            if gap < 0:
                print(instance.name, min(energies))
            gaps[(instance.name, t)] = gap
            times[(instance.name, t)] = elapsed_discrete
            pbar.update(1)

instancje = [instance.name for instance in instances]

x = np.arange(len(instancje))
width = 0.1
offset = 0.15

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-2, -1, 0, 1, 2]
for idx, s in enumerate(steps):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"{s} kroków")
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
if y_max <1.1:
    y_max = 1.1
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji i różnej ilo
ax.axhline(0, color='black', linestyle='--')
```

```
ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()

fig2, ax2 = plt.subplots(figsize=(12, 6))
for idx, s in enumerate(steps):
    a = [times[(i, s)] for i in instancje]
    bar = ax2.bar(x + offset * param[idx], a, width, label=rf"{s} kroków")

ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instanc
ax2.legend()

plt.grid(axis="y")
plt.show()
```

```
zbieranie danych: 100%|██████████| 20/20 [00:17<00:00,  1.17it/s]
```





Bibliografia

- Goto, H., *et al.*, Combinatorial optimization by simulating adiabatic bifurcations in nonlinear Hamiltonian systems. *Sci. Adv.* **5**, eaav2372 (2019). DOI:10.1126/sciadv.aav2372

- Goto, H., *et al.*, High-performance combinatorial optimization based on classical mechanics. *Sci. Adv.* **7**, eabe7953 (2021). DOI:10.1126/sciadv.abe7953

- Pawlowski, J., *et al.* "Closing the Quantum-Classical Scaling Gap in Approximate Optimization." *arXiv preprint* arXiv:2505.22514 (2025). DOI:10.48550/arXiv.2505.22514

Wyżarzanie równoległe

Algorytm **parallel annealing** to współczesna metoda optymalizacji inspirowana ideą **wyżarzania kwantowego**. Został zaproponowany przez Jiang *et al.* w 2023 roku.

Model Isinga

Przyjmujemy następującą konwencję dla modelu Isinga, w której macierz sprzężeń $\bm{J}$ jest symetryczna. Hamiltonian ma postać:

$$ H_{\mathrm{Ising}} = -\sum_{i<j} J_{ij}\sigma_i \sigma_j -\sum_i h_i \sigma_i = -\frac{1}{2}\bm{\sigma}^T \bm{J} \bm{\sigma} -\bm{h}^T \bm{\sigma}. $$

Celem algorytmu jest znalezienie takiej konfiguracji spinów $\bm{\sigma}$, dla której energia H_{Ising} jest jak najmniejsza.

Inspirując się wyżarzaniem kwantowym, rozważamy Hamiltonian układu w postaci:

$$ H_{\mathrm{system}} = \lambda(t)\, H_{\mathrm{initial}} + H_{\mathrm{Ising}}. $$

Składnik H_{initial} pełni rolę członu pomocniczego, który stabilizuje dynamikę na początku działania algorytmu. Przyjmujemy:

$$ H_{\mathrm{initial}} = \frac{1}{2}\sum_i x_i^2. $$

Wektor $\bm{x}$ zawiera tzw. **analogowe spiny**. W przeciwieństwie do spinów Isinga, które przyjmują tylko wartości -1 lub 1, zmienne x_i są ciągłe. Dzięki temu algorytm może stopniowo dochodzić do rozwiązania, zamiast od razu pracować na zmiennych binarnych.

Parametr $\lambda(t)$ zmienia się w czasie. Na początku jego wpływ jest większy, co pomaga ustabilizować układ, a następnie stopniowo maleje, tak aby coraz większe znaczenie miała właściwa energia problemu, czyli H_{Ising}.

Gradient Hamiltonianu układu ma postać:

$$ \nabla H_{\mathrm{system}} = \nabla H_{\mathrm{Ising}} + \lambda(t)\nabla H_{\mathrm{initial}} = -\bm{J}\bm{\sigma} - \bm{h} + \lambda(t)\bm{x}. $$

Wzór ten można interpretować następująco:

- składnik $-\bm{J}\bm{\sigma} - \bm{h}$ „ciągnie” układ w stronę minimum energii problemu Isinga,
- składnik $\lambda(t)\bm{x}$ działa jak regularyzacja i kontroluje przebieg dynamiki.

wartości spinów σ_i są otrzymywane z analogowych spinów poprzez rzutowanie za pomocą funkcji znaku:

$$ \sigma_i = \operatorname{sign}(x_i). $$

Po każdej iteracji wartości analogowych spinów są obcinane do przedziału $[-1,1]$:

$$ \operatorname{clip}(x,-1,1)=\max(-1,\min(1,x)). $$

Operacja ta zapobiega sytuacji, w której wartości x_i rosłyby bez ograniczeń. Ma to znaczenie praktyczne, ponieważ bardzo duże wartości nie poprawiają binaryzacji, a mogą destabilizować działanie algorytmu.

Algorytm aktualizuje zmienne za pomocą dwóch równań:

$$ m(t+1)=\beta\, m(t)-\alpha\, \nabla H_{\mathrm{system}}, $$

$$ x(t+1)=x(t)+m(t+1). $$

Tutaj:

- α jest krokiem gradientu,
- β odpowiada za bezwładność (momentum),
- $m(t)$ jest zmienną pomocniczą opisującą „pęd” układu.

Interpretacja jest podobna jak w metodach gradientowych z momentum:

1. najpierw obliczamy kierunek ruchu na podstawie gradientu,
2. następnie aktualizujemy zmienną pędu m,
3. na końcu przesuwamy pseudospiny $\bm{x}$.

Dzięki składnikowi momentum algorytm nie reaguje wyłącznie na lokalny gradient w jednej iteracji, ale częściowo „pamięta” poprzedni kierunek ruchu. W praktyce może to przyspieszać zbieżność i ułatwiać opuszczanie płytkich minimów lokalnych.

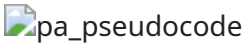
Intuicja działania

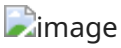
Cały proces można rozumieć następująco:

- na początku układ porusza się w łagodnym krajobrazie energii, kontrolowanym przez człon $\lambda(t)H_{\mathrm{initial}}$,
- z czasem wpływ tego członu maleje,
- coraz silniej ujawnia się właściwy krajobraz energii problemu Isinga,
- zmienne ciągłe $\bm{x}$ stopniowo zbliżają się do wartości odpowiadających spinom binarnym.

W ten sposób algorytm łączy ciągłą dynamikę z problemem dyskretnej optymalizacji.

Pseudokod





In [1]:
```python
# Implementacja Algorytmu dla jednej trajektorii
from setup import setup_paths
setup_paths()

import numpy as np
from typing import Optional
from funkcje_pomocnicze import calculate_energy


def calculate_gradient(J: np.ndarray, h: np.ndarray, x: np.ndarray, state: r
    return -1 * state @ J - h + lambda_t * x


def parrarel_annealing(J, h, step_size: float, lambda_t_max: float, num_step
                       schedule: Optional[np.ndarray] = None, schedule_endpo
    n = len(h)
    x = np.zeros(n)  # stan podstawowy dla H_innit = sum(x**2)
    momentum = np.zeros(n)
    state = np.random.choice([-1, 1], size=n) # losowy stan początkowy

    if schedule is None:
        schedule = np.linspace(lambda_t_max, schedule_endpoint, num=num_step

    for k in range(num_steps):
        lambda_t = schedule[k]
        gradient = calculate_gradient(J, h, x, state, lambda_t)
        momentum = (1 - step_size) * momentum - step_size * gradient
        momentum = np.clip(momentum, -1, 1)
        x += momentum
        x = np.clip(x, -1, 1)
        state = np.sign(x)

    return state, calculate_energy(J, h, state)
```

In [2]:
```python
# Testujemy algorytm na instancji z 216 spinów i z geometryczną strukturą po

import os
import numpy as np

from funkcje_pomocnicze import read_instance, test_pegasus


J, h = read_instance(test_pegasus.path, convention="minus_half")
```

```
state, energy = parrarel_annealing(J, h, step_size=0.01, lambda_t_max=10, nu


print(f"Otrzymana energia: {energy}")
print(f"Stan podstawowy: {test_pegasus.best_energy}")
```

```
Otrzymana energia: -455.0
Stan podstawowy: -469.0
```

Zrównoleglenie

Przez to że algorytm sam w sobie jest stochastyczny nie mamy gwarancji że w trakcie jednego wywołania otrzymamy najlepszy możliwy wynik. Jedym z rozwiązać tego problemu jest zrównoleglenie algorytmu. Oznacza to, że jednocześnie śledzimy wiele "trajektorii" i na końcu obliczeń możemy wybrać najlepszą z nich. Można tego dokonać wywołując algorytm wielokrotnie, ale jest to nieefektywne. Lepszym rozwiązaniem jest trzymanie wielu stanów ułożonych w macierz. Niech σ_i będzie stanem, wtedy mając M trajektorii:

$$ \sigma = \begin{bmatrix} \vert & \vert & \cdots & \vert \\ \sigma_1 & \sigma_2 & \cdots & \sigma_M \\ \vert & \vert & \cdots & \vert \end{bmatrix} $$

Wybór czy trzymamy stany w kolumnach czy w wierszach jest arbitralny. To podejście pozwala nam równolegle liczyć wiele rozwiązań i jest pełną formą tego algorytmu.

W podobny sposób trzymamy x i momentum. Będzie to wymagało pewnych technicznych zmian w kodzie, ale główna idea pozostaje niezmieniona.

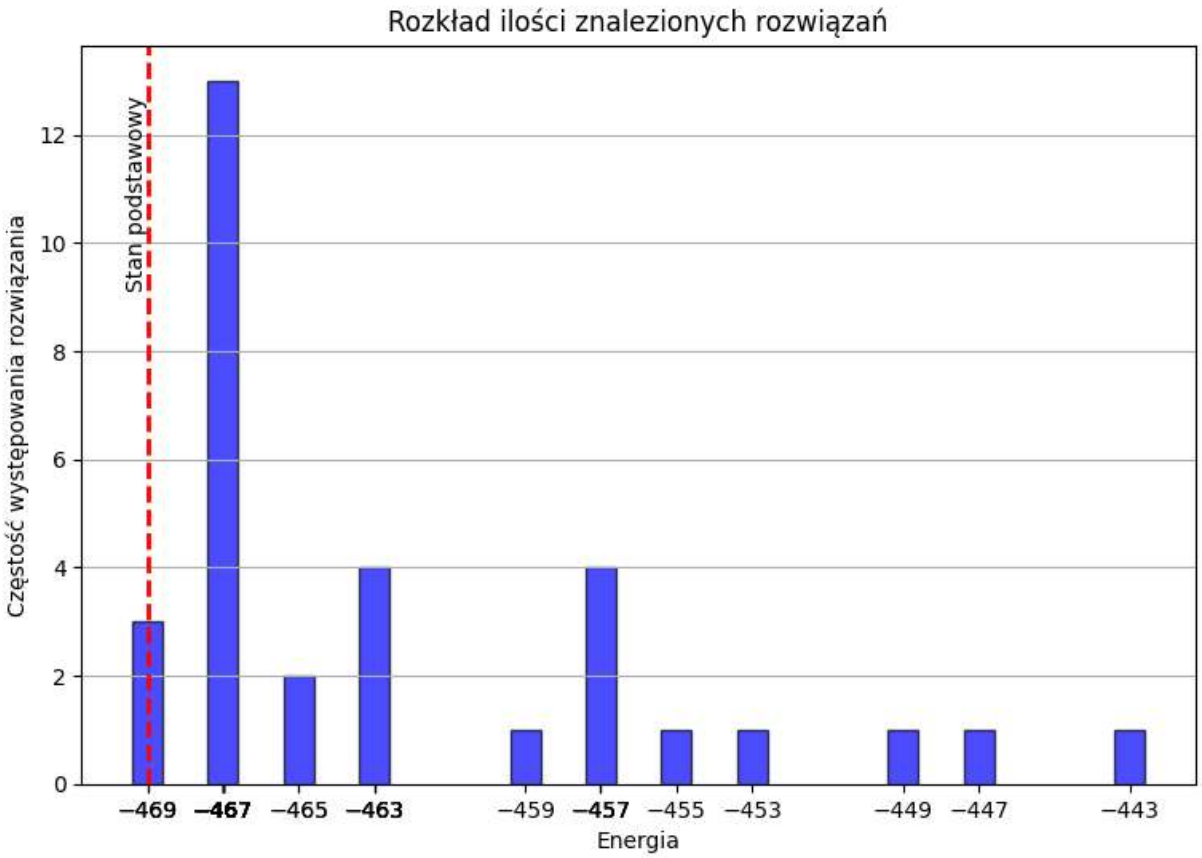
In [3]:
```
# Implementacja wyżarzanie równoległego z wieloma trajektoriami

import numpy as np
from funkcje_pomocnicze import calculate_energy_matrix
from typing import Optional
from tqdm import tqdm


def calculate_gradient_matrix(J: np.ndarray, h: np.ndarray, x: np.ndarray, s
    n = len(h)
    # używamy brodcastingu który jest wykonywany automatycznie w bibliotece
    return -1 * J @ state - h.reshape((n, 1)) + lambda_t * x


def parrarel_annealing_multiple_trajectores(J, h, step_size: float, lambda_t
                        schedule: Optional[np.ndarray] = None, schedule_endpo
    n = len(h)
    x = np.zeros((n, num_trajectories))  # stan podstawowy dla H_innit = sum
    momentum = np.zeros((n, num_trajectories))
    state = np.random.choice([-1, 1], size=(n, num_trajectories))  # losowy
```

```
    if schedule is None:
        schedule = np.linspace(lambda_t_max, schedule_endpoint, num=num_step

    for k in tqdm(range(num_steps), desc="wyżarzanie równoległe"):
        lambda_t = schedule[k]
        gradient = calculate_gradient_matrix(J, h, x, state, lambda_t)
        momentum = (1 - step_size) * momentum - step_size * gradient
        momentum = np.clip(momentum, -1, 1)
        x += momentum
        x = np.clip(x, -1, 1)
        state = np.sign(x)

    return state, calculate_energy_matrix(J, h, state)
```

In [4]:

```
# Ten sam test, wykorzystując wiele trajektorii

import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, test_pegasus


J, h = read_instance(test_pegasus.path, convention="minus_half")

states, energies = parrarel_annealing_multiple_trajectores(J, h, step_size=0



print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {test_pegasus.best_energy}")

unique_values, counts = np.unique(energies, return_counts=True)


plt.figure(figsize=(9, 6))
plt.bar(unique_values, counts, color='blue', edgecolor='black', alpha=0.7)
plt.xticks(energies)
plt.axvline(x=test_pegasus.best_energy, color='red', linestyle='--', linewic
plt.text(test_pegasus.best_energy, plt.ylim()[1]*0.8, 'Stan podstawowy', col
         verticalalignment='center', horizontalalignment='right')

plt.xlabel("Energia")
plt.ylabel("Częstość występowania rozwiązania")
plt.title("Rozkład ilości znalezionych rozwiązań")
plt.grid(axis="y")
plt.show()
```

```
wyżarzanie równoległe: 100%|██████████| 1000/1000 [00:00<00:00, 6071.86it/s]
Otrzymana energia: -469.0
Stan podstawowy: -469.0
```



Z tego wykresu widać, że dla testowanej instancji ten algorytm ma mocną tendencje by utykać w optimum lokalnym, które w tym wypadku jest pierwszym stanem wzbudzonym.

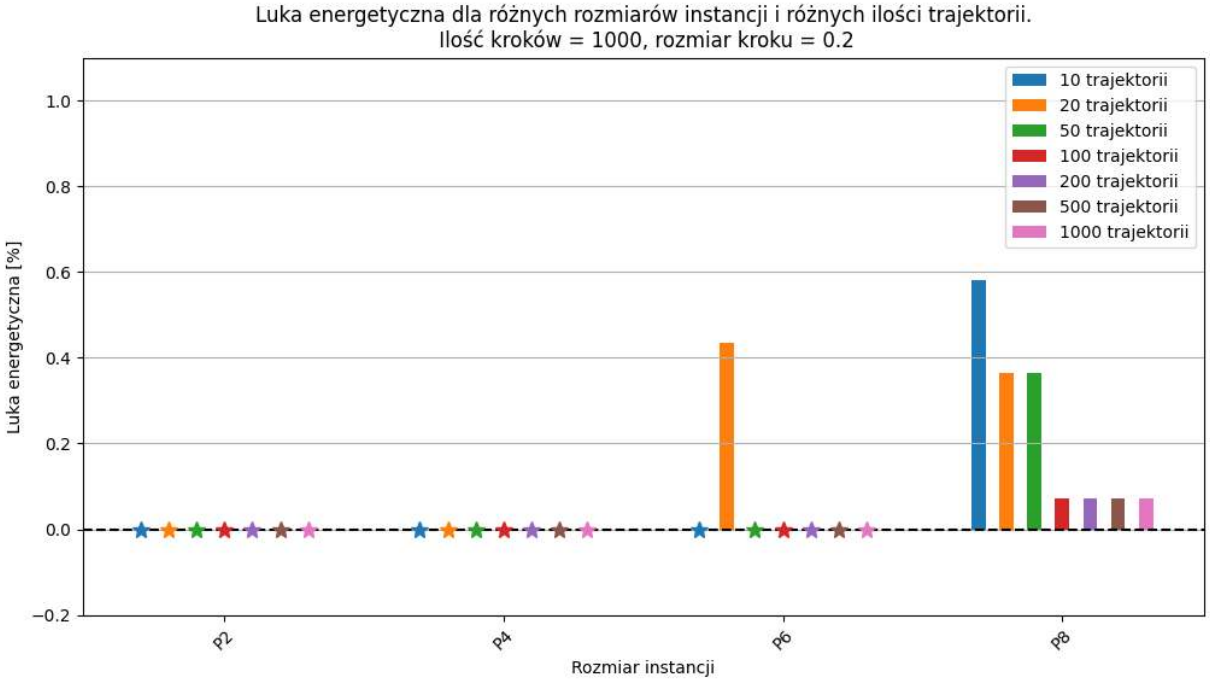
Zachowanie algorytmu dla różnych wartości parametrów.

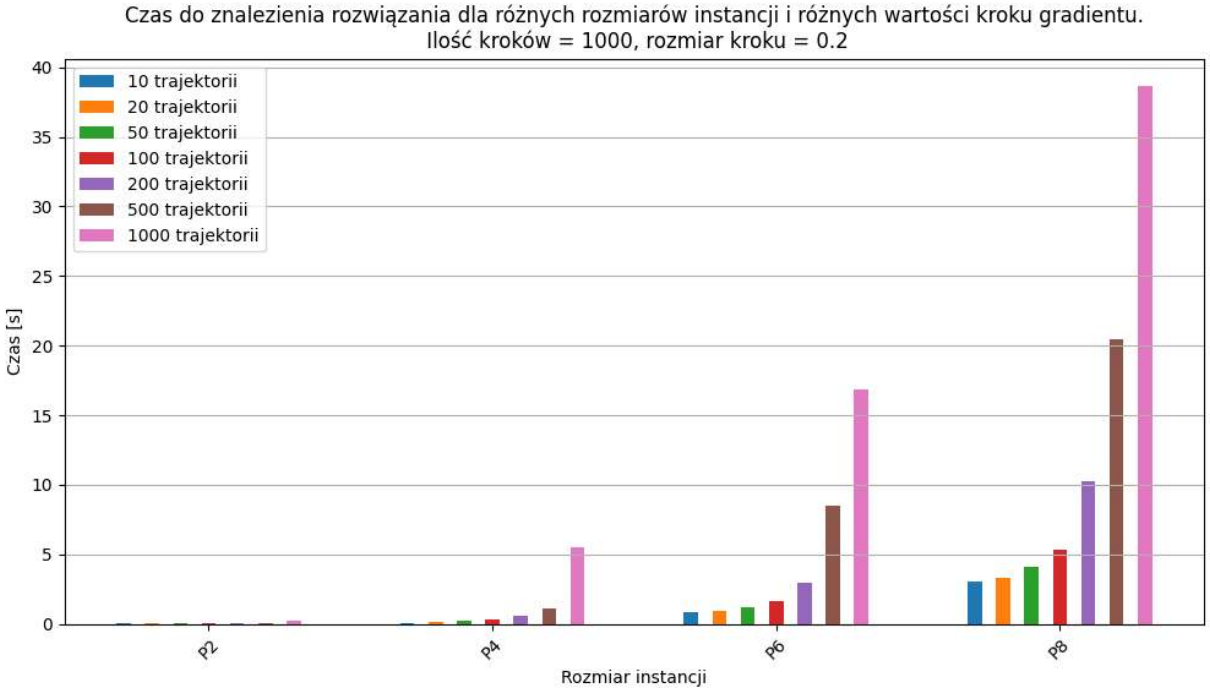
In [5]:
```
# zachowanie względem ilości trajektorii
import time
import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8
from tqdm import tqdm
from IPython.utils.io import capture_output

trajs = [10, 20, 50, 100, 200, 500, 1000]
gaps = {}
times = {}
with tqdm(total=4*len(trajs), desc="zbieranie danych") as pbar:
    for instance in [P2, P4, P6, P8]:
        for num_traj in trajs:
            J, h = read_instance(instance.path, convention="minus_half")
            with capture_output() as captured:
                start = time.time()
                states, energies = parrarel_annealing_multiple_trajectores(J
                                                                           r
```

```
                end = time.time()
            energy = min(energies)
            gap = (instance.best_energy - energy)/instance.best_energy
            gaps[(instance.name, num_traj)] = gap * 100
            times[(instance.name, num_traj)] = end - start
            pbar.update(1)


instancje = ["P2", "P4", "P6", "P8"]

x = np.arange(len(instancje))
width = 0.05
offset = 0.1

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-3, -2, -1, 0, 1, 2, 3]
for idx, s in enumerate(trajs):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"{s} trajektorii
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
if y_max < 1.1:
    y_max = 1.1
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji i różnych il
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()

fig2, ax2 = plt.subplots(figsize=(12, 6))
for idx, s in enumerate(trajs):
    a = [times[(i, s)] for i in instancje]
    bar = ax2.bar(x + offset * param[idx], a, width, label=rf"{s} trajektori

ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instanc
ax2.legend()
```

```
plt.grid(axis="y")
plt.show()
```

```
zbieranie danych: 100%|██████████| 28/28 [02:08<00:00,  4.59s/it]
```



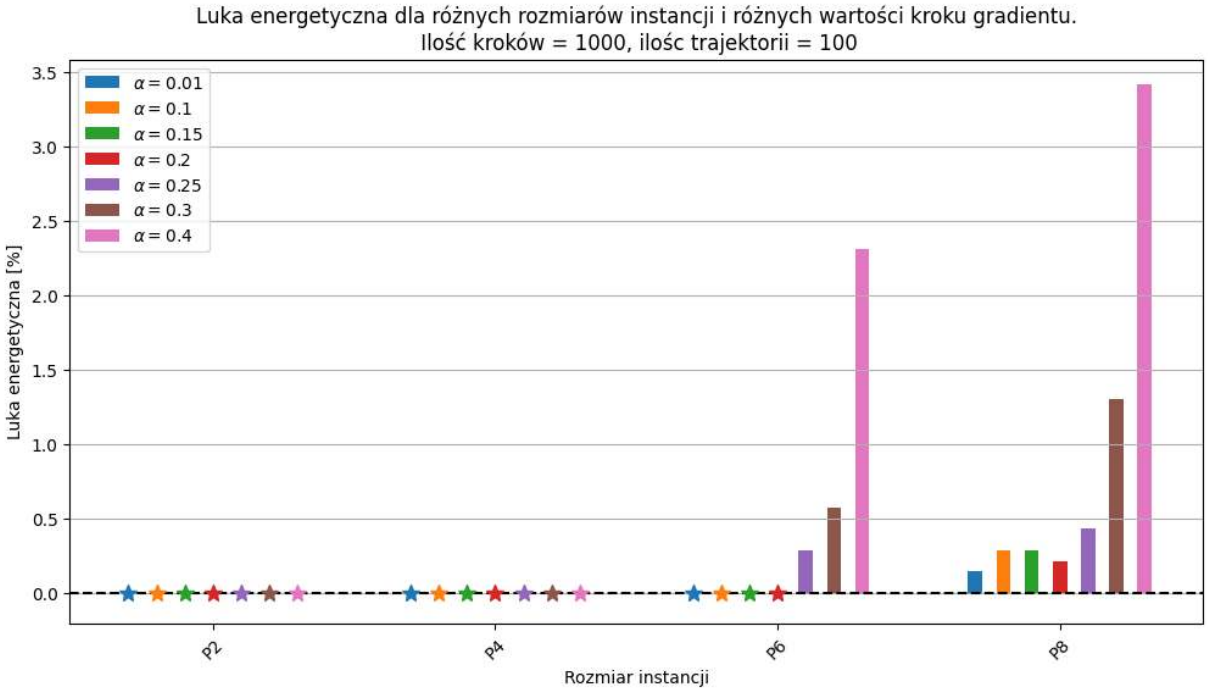


In [6]:
```
# zachowanie względem rozmiaru kroku w gradiencie

import time
import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, P2, P4, P6, P8
from tqdm import tqdm
from IPython.utils.io import capture_output
```

```
stepsizes = [0.01, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4]
gaps = {}
times = {}
with tqdm(total=4*len(stepsizes), desc="zbieranie danych") as pbar:
    for instance in [P2, P4, P6, P8]:
        for step_size in stepsizes:
            J, h = read_instance(instance.path, convention="minus_half")
            with capture_output() as captured:
                start = time.time()
                states, energies = parrarel_annealing_multiple_trajectores(J
                                                                           r
                end = time.time()
            energy = min(energies)
            gap = (instance.best_energy - energy)/instance.best_energy
            if gap < 0:
                print(instance.name, energy)
            gaps[(instance.name, step_size)] = gap * 100
            times[(instance.name, step_size)] = end - start
            pbar.update(1)


instancje = ["P2", "P4", "P6", "P8"]

x = np.arange(len(instancje))
width = 0.05
offset = 0.1

fig, ax = plt.subplots(figsize=(12, 6))
barset = []
param = [-3, -2, -1, 0, 1, 2, 3]
for idx, s in enumerate(stepsizes):
    a = [gaps[(i, s)] for i in instancje]
    bar = ax.bar(x + offset * param[idx], a, width, label=rf"$\alpha={s}$")
    barset.append(bar)


(y_min, y_max) = ax.get_ylim()
ax.set_ylim((-0.2, y_max))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji i różnych wa
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in barset:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.get_face
plt.grid(axis="y")
plt.show()
```

```
# fig2, ax2 = plt.subplots(figsize=(12, 6))
# for idx, s in enumerate(stepsizes):
#     a = [times[(i, s)] for i in instancje]
#     bar = ax2.bar(x + offset * param[idx], a, width, label=rf"$\alpha={s}$

# ax2.set_xlabel("Rozmiar instancji")
# ax2.set_xticks(x)
# ax2.set_xticklabels(instancje, rotation=45)
# ax2.set_ylabel("Czas [s]")
# ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów insta
# ax2.legend()

# plt.grid(axis="y")
# plt.show()
```

```
zbieranie danych: 100%|██████████| 28/28 [00:56<00:00,  2.00s/it]
```



Bibliografia

- Jiang, M., Shan, K., He, C. *et al.* Efficient combinatorial optimization by quantum-inspired parallel annealing in analogue memristor crossbar. *Nat Commun* **14**, 5927 (2023). DOI: 10.1038/s41467-023-41647-2

05_Algorytm_heurystycznego_branch_and_bound

May 26, 2026

1 Branch and Bound

Algorytm branch and bound polega na systematycznym przeszukiwaniu przestrzeni rozwiązań problemu optymalizacji kombinatorycznej przy jednoczesnym ograniczaniu liczby badanych przypadków. W ogólności, składa się z 3 rodzajów operacji:

- **Rozgałęzianie (branching)**: Problem jest dzielony na mniejsze podproblemy, które odpowiadają różnym podzbiorom przestrzeni rozwiązań.
- **Wyznaczanie ograniczeń (bounding)**: Dla każdego podproblemu oblicza się ograniczenie (np. dolne lub górne), które informuje o tym, jaką najlepszą wartość funkcji celu można osiągnąć w obrębie danego poddrzewa.
- **Odrzucanie niekorzystnych gałęzi (pruning)**: Jeśli obliczone ograniczenie dla danego podproblemu nie pozwala na uzyskanie lepszego wyniku niż już znalezione rozwiązanie, cały podproblem jest pomijany, co znacząco zmniejsza przestrzeń poszukiwań.

1.1 Branch and Bound dla modelu Isinga.

Pokażemy heurystyczną wersję tego algorytmu dla modelu Isinga. Nie daje nam gwarancji znalezienia stanu podstawowego. Zeletą jest łatwośc implementacji. Jest to punk wyjścia do budowania innych algorytmów o podobnej strukturze (np. branch and bound w przestrzeni probabilistycznej.)

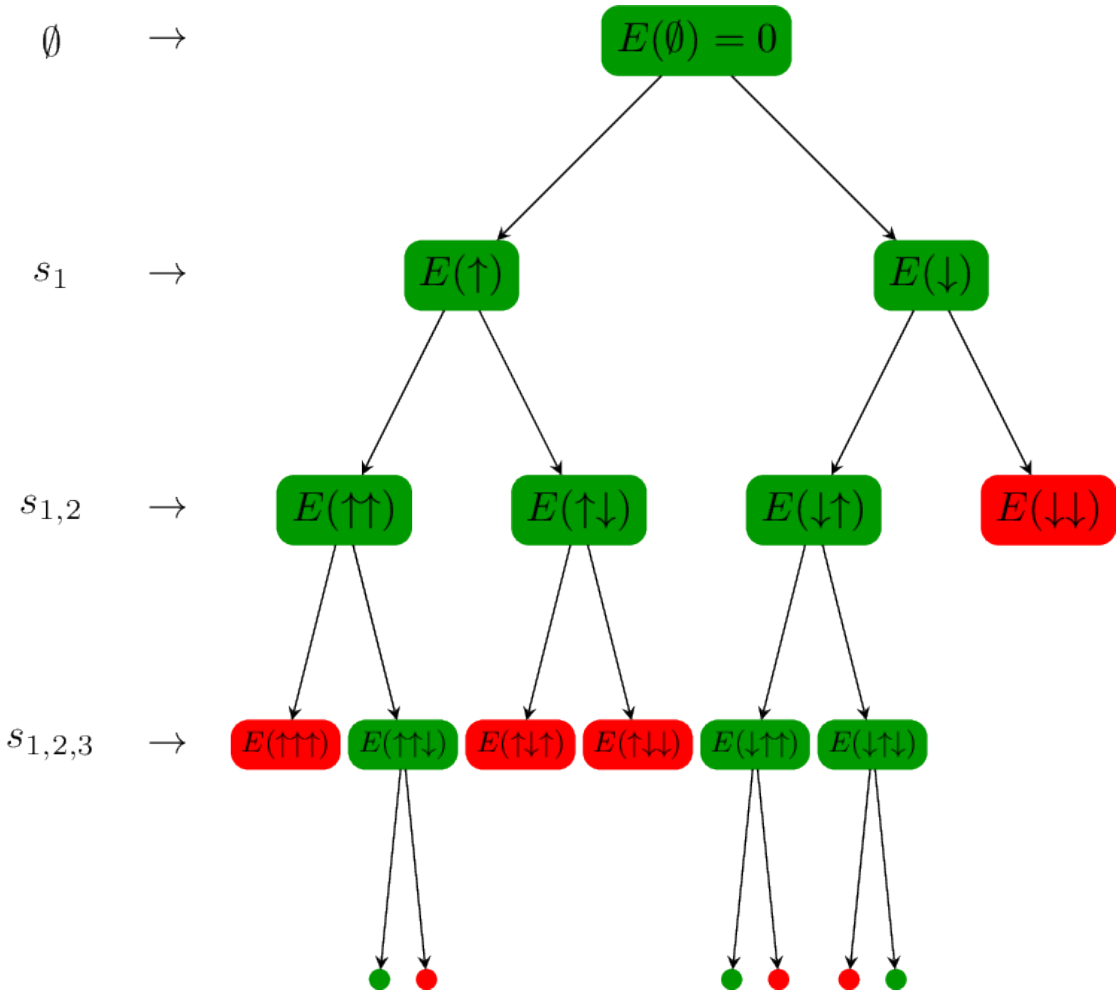
Pseudokod wygląda w następująco:

Algorithm 1: Heuristic Branch-and-Bound for Ising model

Data: Number of retained solutions M, Number of spins N
Result: Approximate low energy spectrum of given problem

$S \leftarrow \{\emptyset\}$;
$c \leftarrow 0$; `// counter`
while $c \leq N$ **do**
    `// Branching`
    for $s \in S$ **do**
        $s_+ \leftarrow s+ \uparrow$;
        $s_- \leftarrow s+ \downarrow$;
        $S \leftarrow S \cup \{s_+, s_-\}$
    end
    `// Bounding`
    if $|S| > M$ **then**
        `// remove solutions with largest energies until` M `solutions are left`
        `sort(`S`)`;
        $S \leftarrow S[1 : M]$;
    end
    $c \leftarrow c + 1$
end
return S

Poniżej jest graficzna reprezentacja jak takie przeszukiwanie drzewa może wyglądać.



[2]:
```python
# Ustawienie ścierzek dla plików pomocniczych
from setup import setup_paths
setup_paths()

# Implementacja prostego Branch and Bound
import numpy as np
from typing import NamedTuple
from math import inf
from tqdm import tqdm


class State(NamedTuple):
    configuration: list
    energy: float


# Funkcja pomocniczna, warto zauważyć że są prowadzone pewne modyfikacje
def calculate_energy(J: np.ndarray, h: np.ndarray, state: list):
```

```
    n = len(h)
    k = len(state)
    state = state + [0] * (n - k)
    state = np.array(state)
    return state @ J @ state.T + state @ h


def branch_and_bound(J, h, num_states, num_returned_elements = 1):

    n = len(h)
    states = [State([], inf)]

    for _ in tqdm(range(n), desc="Wykonywanie branch and bound"):
        # Branching
        temp = []
        for state in states:
            plus = state.configuration + [1]
            plus_energy = calculate_energy(J, h, plus)
            temp.append(State(plus, plus_energy))

            minus = state.configuration + [-1]
            minus_energy = calculate_energy(J, h, minus)
            temp.append(State(minus, minus_energy))

        # Bounding
        temp.sort(key=lambda x: x.energy)
        if len(temp) > num_states:
            temp = temp[:num_states]
        states = temp

    return states[:num_returned_elements]
```

[3]:
```
# test dla bardzo małej instancji (24 spiny)

from funkcje_pomocnicze import read_instance, small_pegasus

J, h = read_instance(small_pegasus.path, convention="dwave")



solutions = branch_and_bound(J, h, 100)
solution = solutions[0]

print(f"Otrzymana energia: {solution.energy}")
print(f"Stan podstawowy: {small_pegasus.best_energy}")
```

```
Wykonywanie branch and bound: 100%|      | 24/24 [00:00<00:00, 915.06it/s]

Otrzymana energia: -39.0
Stan podstawowy: -39.0
```

1.2 Testy algorytmu

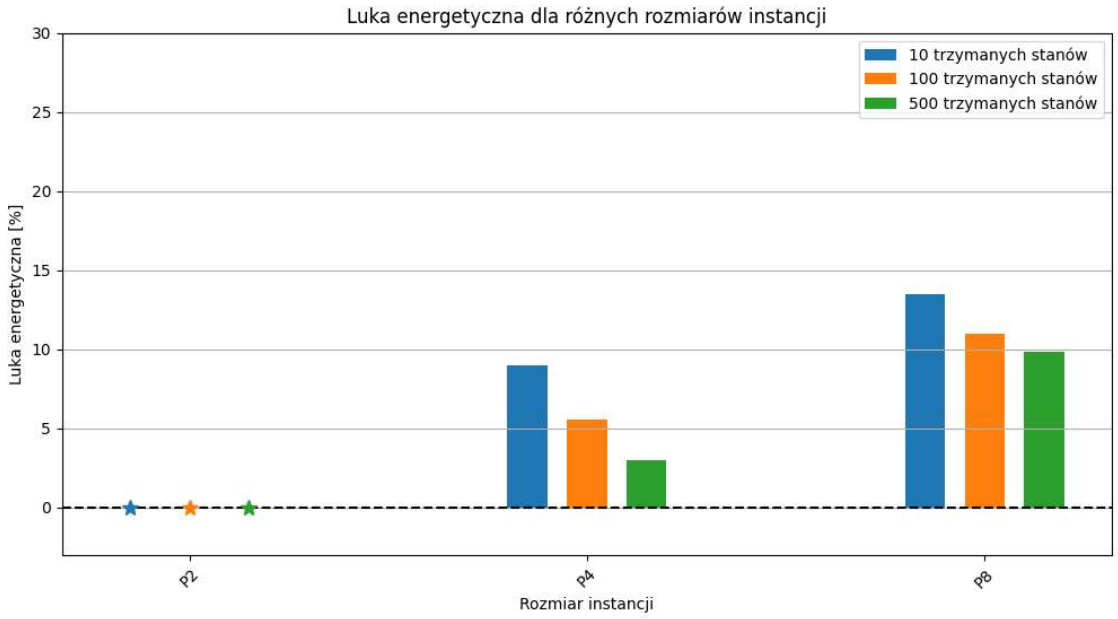
W następnych komurkach przeprowadzimy testy jak ten algorytm się zachowuje dla różnych rozmiarów instancji oraz różnych wartości

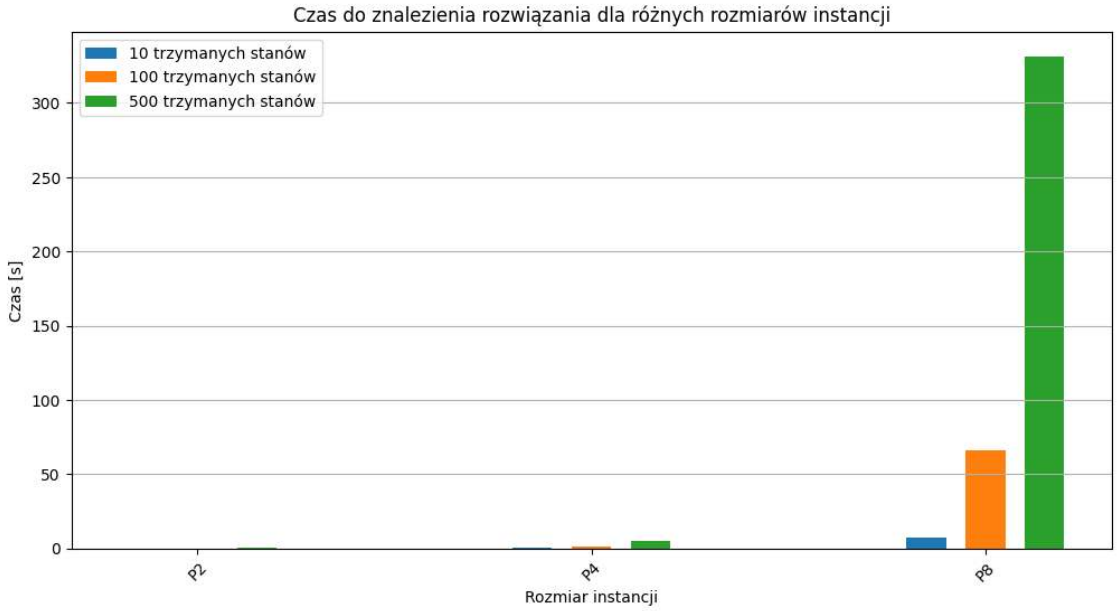
[ ]:
```python
# wykresy zachowania algorytmu

import os
import time
import matplotlib.pyplot as plt
import numpy as np

from Setup import setup
setup()
from funkcje_pomocnicze import P2, P4, P8, read_instance
from IPython.utils.io import capture_output
from tqdm import tqdm

gaps = {}
times = {}

is_automated_run = "PYTEST_CURRENT_TEST" in os.environ
instances = [P2, P4] if is_automated_run else [P2, P4, P8]
beam_widths = [10, 100] if is_automated_run else [10, 100, 500]

with tqdm(total=len(instances) * len(beam_widths), desc="zbieranie danych") as␣
 ↪pbar:
    for instance in instances:
        for m in beam_widths:
            J, h = read_instance(instance.path, convention="dwave")
            with capture_output() as captured:
                start = time.time()
                solutions = branch_and_bound(J, h, m)
                solution = solutions[0]
                end = time.time()
            gap = (instance.best_energy - solution.energy)/instance.best_energy
            gaps[(instance.name, m)] = gap * 100
            times[(instance.name, m)] = end - start
            pbar.update(1)


instancje = [instance.name for instance in instances]
```

```
x = np.arange(len(instancje))
width = min(0.8 / max(len(beam_widths), 1), 0.25)
offsets = np.linspace(-width * (len(beam_widths) - 1) / 2, width *␣
 ↪(len(beam_widths) - 1) / 2, len(beam_widths))

fig, ax = plt.subplots(figsize=(12, 6))
bars_by_width = []
for offset, m in zip(offsets, beam_widths):
    values = [gaps[(instance_name, m)] for instance_name in instancje]
    bars = ax.bar(x + offset, values, width, label=f"{m} trzymanych stanów")
    bars_by_width.append(bars)

ax.set_ylim((-3, 30))
ax.set_xlabel("Rozmiar instancji")
ax.set_xticks(x)
ax.set_xticklabels(instancje, rotation=45)
ax.set_ylabel("Luka energetyczna [%]")
ax.set_title("Luka energetyczna dla różnych rozmiarów instancji")
ax.axhline(0, color='black', linestyle='--')

ax.legend()

for bars in bars_by_width:
    for bar in bars:
        if bar.get_height() == 0:
            center = bar.get_x() + bar.get_width()/2
            ax.plot(center, 0, marker='*', markersize=10, color=bar.
 ↪get_facecolor())
plt.grid(axis="y")
plt.show()


fig2, ax2 = plt.subplots(figsize=(12, 6))
for offset, m in zip(offsets, beam_widths):
    values = [times[(instance_name, m)] for instance_name in instancje]
    ax2.bar(x + offset, values, width, label=f"{m} trzymanych stanów")

ax2.set_xlabel("Rozmiar instancji")
ax2.set_xticks(x)
ax2.set_xticklabels(instancje, rotation=45)
ax2.set_ylabel("Czas [s]")
ax2.set_title("Czas do znalezienia rozwiązania dla różnych rozmiarów instancji")
ax2.legend()

plt.grid(axis="y")
```

```
plt.show()
```

```
zbieranie danych: 100%|      | 9/9 [06:50<00:00, 45.64s/it]
```





06_GPU

May 26, 2026

1 Wykorzystanie procesorów graficznych (GPU) w algorytmach heurystycznych

1.1 Wstęp - jak używać GPU

Podstawową zaletą używania GPU jest duża możliwość równoległego wykonywania obliczeń, co pozwala znacząco przyspieszyć operacje wymagające intensywnej mocy obliczeniowej. W praktyce istnieją dwa główne sposoby wykorzystania GPU: można sięgnąć po gotowe implementacje dostępne w popularnych bibliotekach i frameworkach (np. TensorFlow, PyTorch czy CuPy), które automatycznie obsługują akcelerację na karcie graficznej, lub napisać własne jądro CUDA, co daje pełną kontrolę nad strukturą obliczeń i umożliwia optymalizację pod kątem konkretnego zadania.

Pisanie własnych jąder CUDA wymaga pewnej wprawy, ale pozwala na uzyskanie maksymalnej wydajności w niestandardowych zastosowaniach. Pardzo dobry poradnik jak to robić mozna znaleść na stronie CUDA C++ Programming Guide.

2 Symulowana bifurkacja

[1]:
```
# Balistyczna symulowana bifurkacja na gpu naiwna - zamieniamy np na cp

from setup import setup_paths
setup_paths()


import cupy as cp
from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze_gpu import calculate_energy_gpu


def wall_gpu(x: cp.ndarray, y: cp.ndarray):
    mask = cp.abs(x) > 1
    x[mask] = cp.sign(x[mask])
    y[mask] = 0
    return x, y
```

```
def balistic_simulated_bifurcation_gpu_naive(J, h, num_steps, time_step,␣
 ↪num_trajectories: int,
                                    a_0: Optional[float] = None, c_0_scaling:␣
 ↪Optional[float] = None):
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = cp.sqrt(cp.sum(cp.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling


    a = cp.linspace(0, a_0, num=num_steps)

    x = cp.zeros((N, num_trajectories))
    y = cp.random.uniform(-0.1, 0.1, (N, num_trajectories))

    for t in tqdm(range(num_steps), desc="symulowana bifurkacja"):
        y += (-1 * (a_0 - a[t]) * x + c_0 * (cp.matmul(J,x) + h.reshape((N,␣
 ↪1)))) * time_step  # x(t)
        x += a_0 * y * time_step # x(t + 1)

        x, y = wall_gpu(x, y)

    x = cp.sign(x)
    return x, calculate_energy_gpu(J, h, x)
```

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, test_pegasus, small_pegasus, P12

instance = P12

J, h = read_instance(instance.path, convention="minus_half")

n = len(h)
print(n)

J = cp.asarray(J, dtype=cp.float32)
h = cp.asarray(h, dtype=cp.float32)

states, energies = balistic_simulated_bifurcation_gpu_naive(J, h, time_step=0.
 ↪5, num_steps=200, num_trajectories=2000, c_0_scaling=0.7)
```

```
# Konwersja z karty graficznej na CPU
energies = cp.asnumpy(energies)
states = cp.asnumpy(states)

print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/(2 *␣
 ↪instance.best_energy) * 100):.2f}%")

unique_values, counts = np.unique(energies, return_counts=True)
```

```
2904
```

```
symulowana bifurkacja: 100%|      | 200/200 [00:13<00:00, 14.77it/s]
```

```
Otrzymana energia: -6713.0
Stan podstawowy: -6831.0
Luka energetyczna: 0.86%
```

[6]:
```
# Implementacja SBM używając własnego kernelu

from setup import setup_paths
setup_paths()


import cupy as cp
from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze_gpu import calculate_energy_gpu

def balistic_simulated_bifurcation_gpu(J, h, num_steps, time_step,␣
 ↪num_trajectories: int,
                                       a_0: Optional[float] = None, c_0_scaling:␣
 ↪Optional[float] = None):
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = np.sqrt(np.sum(np.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling

    dtype = cp.float32
```

```
    a = [dtype(i * a_0 / (num_steps - 1)) for i in range(num_steps)]

    a_0 = dtype(a_0)
    c_0 = dtype(c_0.item())
    time_step = dtype(time_step)

    x = cp.zeros((N, num_trajectories), dtype=dtype)
    y = cp.random.uniform(-0.1, 0.1, (N, num_trajectories), dtype=dtype)

    x_new = cp.empty_like(x)
    y_new = cp.empty_like(y)

    threadsperblock = 256  # Ilość wątków w bloku,
    blockspergrid_x = num_trajectories  # każdy blok zajmuje się trajektorią
    blockspergrid_y = (N + threadsperblock - 1) // threadsperblock  #␣
↪wystarczająca ilość bloków by pomieścić całą kolumnę
    blockspergrid = (blockspergrid_x, blockspergrid_y)

    kernel = cp.RawModule(path="cuda_kernels/sbm_kernel.ptx")
    update_y = kernel.get_function("update_y")
    update_x_and_wall = kernel.get_function("update_x_and_wall")

    for t in tqdm(range(num_steps), desc="balistyczna symulowana bifurkacja"):
        A = cp.matmul(J, x) + h.reshape((N, 1))
        update_y(blockspergrid, (threadsperblock,),(x, y, A,
                                                    a_0, a[t], c_0, time_step,␣
↪N,
                                                    y_new))
        y = y_new
        update_x_and_wall(blockspergrid, (threadsperblock,),(x, y,
                                                             a_0, time_step, N,
                                                             x_new, y_new))
        x = x_new
        y = y_new

    x = cp.sign(x)
    return x, calculate_energy_gpu(J, h, x)
```

[10]:
```
# testy dla większej instancji

import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, test_pegasus, small_pegasus, P12

instance = P12
```

```
J, h = read_instance(instance.path, convention="minus_half")

J = cp.asarray(J, dtype=cp.float32)
h = cp.asarray(h, dtype=cp.float32)

states, energies = balistic_simulated_bifurcation_gpu(J, h, time_step=0.5,␣
 ↪num_steps=200, num_trajectories=2000, c_0_scaling=0.7)

# Konwersja z karty graficznej na CPU
energies = cp.asnumpy(energies)
states = cp.asnumpy(states)

print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/(2 *␣
 ↪instance.best_energy) * 100):.2f}%")
```

```
balistyczna symulowana bifurkacja: 100%|      | 200/200 [00:00<00:00,
6287.65it/s]

Otrzymana energia: -6713.0
Stan podstawowy: -6831.0
Luka energetyczna: 0.86%
```

[21]:
```
# Dyskretna symulowana bifurkacja na gpu naiwna - zamieniamy np na cp

from setup import setup_paths
setup_paths()


import cupy as cp
from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze_gpu import calculate_energy_gpu


def wall_gpu(x: cp.ndarray, y: cp.ndarray):
    mask = cp.abs(x) > 1
    x[mask] = cp.sign(x[mask])
    y[mask] = 0
    return x, y


def discrete_simulated_bifurcation_gpu_naive(J, h, num_steps, time_step,␣
 ↪num_trajectories: int,
```

```
                                    a_0: Optional[float] = None, c_0_scaling:␣
↪Optional[float] = None):
    if a_0 is None:
        a_0 = 1

    N, _ = J.shape
    mean_J = cp.sqrt(cp.sum(cp.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling


    a = cp.linspace(0, a_0, num=num_steps)

    x = cp.zeros((N, num_trajectories))
    y = cp.random.uniform(-0.1, 0.1, (N, num_trajectories))

    for t in tqdm(range(num_steps), desc="symulowana bifurkacja"):
        y += (-1 * (a_0 - a[t]) * x + c_0 * (cp.matmul(J,cp.sign(x)) + h.
↪reshape((N, 1)))) * time_step  # x(t)
        x += a_0 * y * time_step # x(t + 1)

        x, y = wall_gpu(x, y)

    x = cp.sign(x)
    return x, calculate_energy_gpu(J, h, x)
```

[22]:
```
# Dyskretna Symulowana Bifurkacja - własny kernel


from setup import setup_paths
setup_paths()

import cupy as cp
from math import sqrt
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze_gpu import calculate_energy_gpu


def discrete_simulated_bifurcation_gpu(J, h, num_steps, time_step,␣
↪num_trajectories: int,
                                    a_0: Optional[float] = None, c_0_scaling:␣
↪Optional[float] = None):
    if a_0 is None:
```

```
        a_0 = 1

    N, _ = J.shape
    mean_J = cp.sqrt(cp.sum(cp.square(J)) / (N * (N - 1)) )
    c_0 = 0.5 / (mean_J * sqrt(N))

    if c_0_scaling is not None:
        c_0 *= c_0_scaling

    dtype = cp.float32

    a = [dtype(i * a_0 / (num_steps - 1)) for i in range(num_steps)]

    a_0 = dtype(a_0)
    c_0 = dtype(c_0.item())
    time_step = dtype(time_step)

    x = cp.zeros((N, num_trajectories), dtype=dtype)
    y = cp.random.uniform(-0.1, 0.1, (N, num_trajectories), dtype=dtype)

    x_new = cp.empty_like(x)
    y_new = cp.empty_like(y)

    threadsperblock = 256  # Ilość wątków w bloku,
    blockspergrid_x = num_trajectories  # każdy blok zajmuje się trajektorią
    blockspergrid_y = (N + threadsperblock - 1) // threadsperblock  #␣
↪wystarczająca ilość bloków by pomieścić całą kolumnę
    blockspergrid = (blockspergrid_x, blockspergrid_y)

    kernel = cp.RawModule(path="cuda_kernels/sbm_kernel.ptx")
    update_y = kernel.get_function("update_y")
    update_x_and_wall = kernel.get_function("update_x_and_wall")

    for t in tqdm(range(num_steps), desc="dyskretna symulowana bifurkacja"):
        sign_x = cp.sign(x)
        A = cp.matmul(J, sign_x) + h.reshape((N, 1))
        update_y(blockspergrid, (threadsperblock,),(x, y, A,
                                                    a_0, a[t], c_0, time_step,␣
↪N,
                                                    y_new))
        y = y_new
        update_x_and_wall(blockspergrid, (threadsperblock,),(x, y,
                                                             a_0, time_step, N,
                                                             x_new, y_new))
        x = x_new
        y = y_new
```

```
    x = cp.sign(x)
    return x, calculate_energy_gpu(J, h, x)
```

[ ]:
```
# testy dla większej instancji

import matplotlib.pyplot as plt
import numpy as np
from funkcje_pomocnicze import read_instance, test_pegasus, small_pegasus, P8

instance = P8

J, h = read_instance(instance.path, convention="minus_half")

J = cp.asarray(J, dtype=cp.float32)
h = cp.asarray(h, dtype=cp.float32)

states, energies = discrete_simulated_bifurcation_gpu(J, h, time_step=0.5,␣
 ↪num_steps=200, num_trajectories=2000, c_0_scaling=0.7)


print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/(2 *␣
 ↪instance.best_energy) * 100):.2f}%")
```

```
dyskretna symulowana bifurkacja: 100%|      | 200/200 [00:00<00:00,
4180.57it/s]

Otrzymana energia: -2740.0
Stan podstawowy: -2752.0
Luka energetyczna: 0.22%
```

2.1 Wyżarzanie równoległe

[24]:
```
# Implementacja algorytmu wyżarzania równoległego używając własnego kernelu w␣
 ↪CUDA
from setup import setup_paths
setup_paths()


import os
import cupy as cp
from typing import Optional
from tqdm import tqdm

from funkcje_pomocnicze_gpu import calculate_energy_gpu
```

```
def parrarel_annealing_gpu(J, h, step_size: float, lambda_t_max: float,␣
↪num_steps: int, num_trajectories: int,
                       schedule: Optional[list] = None, dtype = cp.float32):

    n = len(h)
    x = cp.zeros((n, num_trajectories), dtype=dtype)  # stan podstawowy dla␣
↪H_innit = sum(x**2)
    momentum = cp.zeros((n, num_trajectories), dtype=dtype)
    state = cp.random.choice([dtype(-1.), dtype(1.)], size=(n,␣
↪num_trajectories)) # losowy stan początkowy
    step_size = dtype(step_size)

    if schedule is None:
        schedule = [dtype(lambda_t_max * (1 - i/(num_steps -1))) for i in␣
↪range(num_steps)]  # dlaczego nie linspace? chodzi o typy

    # cu_path = os.path.join("cuda_kernels", "pa_kernel.cu")
    # ptx_path = os.path.join("cuda_kernels", "pa_kernel.ptx")
    # command = f"nvcc --ptx --define-macro N={n} --define-macro␣
↪M={num_trajectories} {cu_path} -o {ptx_path}"

    # os.system(command)
    kernel = cp.RawModule(path="cuda_kernels/pa_kernel.ptx")
    parrarel_annealing_step = kernel.get_function("parrarel_annealing_step")

    threadsperblock = 256  # Ilość wątków w bloku,
    blockspergrid_x = num_trajectories  # każdy blok zajmuje się trajektorią
    blockspergrid_y = (n + threadsperblock - 1) // threadsperblock  #␣
↪wystarczająca ilość bloków by pomieścić całą kolumnę
    blockspergrid = (blockspergrid_x, blockspergrid_y)

    x_new = cp.empty_like(x)
    momentum_new = cp.empty_like(momentum)
    state_new = cp.empty_like(state)

    for k in tqdm(range(num_steps), desc="wyżarzanie równoległe GPU"):

        lambda_t = schedule[k]
        A = cp.matmul(J, state)
        parrarel_annealing_step(blockspergrid, (threadsperblock,), (A, h, x,␣
↪momentum,
                                                                    lambda_t,␣
↪step_size, n,
                                                                    momentum_new,␣
↪x_new, state_new))
```

```
        momentum = momentum_new
        x = x_new
        state = state_new

    return state, calculate_energy_gpu(J, h, state)
```

[25]:
```
from funkcje_pomocnicze import read_instance, small_pegasus, test_pegasus

instance = small_pegasus

J, h = read_instance(instance.path, convention="minus_half")


J = cp.asarray(J, dtype=cp.float32)
h = cp.asarray(h, dtype=cp.float32)

states, energies = parrarel_annealing_gpu(J, h, step_size=0.01,␣
 ↪lambda_t_max=10, num_steps=1000, num_trajectories=5000)
print(f"Otrzymana energia: {min(energies)}")
print(f"Stan podstawowy: {instance.best_energy}")
print(f"Luka energetyczna: {((instance.best_energy - min(energies))/(2 *␣
 ↪instance.best_energy) * 100):.2f}%")
```

```
wyżarzanie równoległe GPU: 100%|      | 1000/1000 [00:00<00:00, 9518.56it/s]

Otrzymana energia: -39.0
Stan podstawowy: -39.0
Luka energetyczna: -0.00%
```

2.2 Bruteforce

Ta implementacja pochodzi z artykułu:

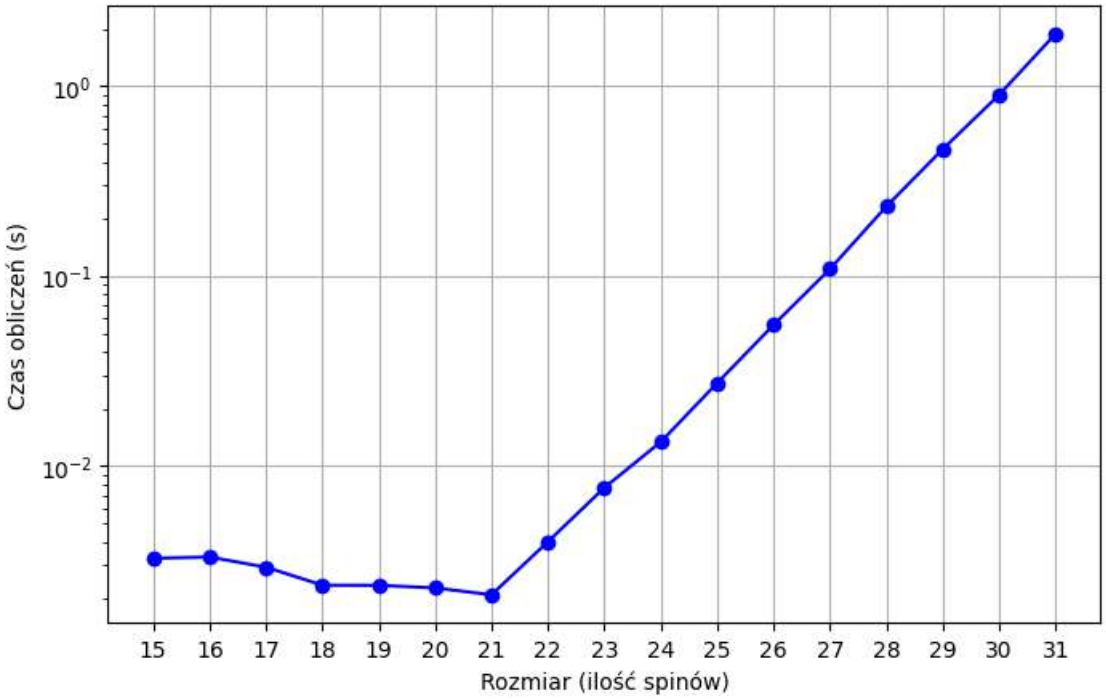
Jałowiecki, K., Rams, M. M., & Gardas, B. (2021). Brute-forcing spin-glass problems with CUDA. *Computer Physics Communications*, *260*, 107728. DOI: 10.1016/j.cpc.2020.107728

[26]:
```
# brute force CUDA

import cupy as cp
from tqdm import tqdm

def select_lowest(energies, states, num_states):

    indices = cp.argpartition(energies, num_states)[:num_states]

    low_energies = energies[indices]
    low_states = states[indices]
    return low_energies, low_states
```

```
def sort_by_key(energies, states, num_states):
    order = cp.argsort(energies)[:num_states]

    low_energies = energies[order]
    low_states = states[order]
    return low_energies, low_states


def brute_force_gpu(Q,  num_states: int, sweep_size_exponent: int = 10,␣
 ↪threadsperblock: int = 256):
    N, _ = Q.shape
    if N > 64:
        raise ValueError("Za wysoka wartość N. Ta implementacja wspiera co␣
 ↪najwyżej 64 spiny (64-bitowy integer)")
    sweep_size = 2**sweep_size_exponent
    num_chunks = 2**(N-sweep_size_exponent)


    brute_force_kernel = cp.RawModule(path="cuda_kernels/brute_force_kernel.
 ↪ptx")
    compute_energies = brute_force_kernel.get_function("compute_energies")

    blockspergrid = sweep_size//threadsperblock

    final_energies = cp.array([])
    final_states = cp.array([])

    for i in tqdm(range(num_chunks), desc="wyczerpujące przeszukiwanie"):

        energies = cp.empty(sweep_size, dtype=cp.float32)
        states = cp.empty(sweep_size, dtype=cp.int64)
        compute_energies((blockspergrid,), (threadsperblock,),
                         (Q, cp.int32(N), cp.int32(sweep_size_exponent),␣
 ↪energies, states, cp.int64(i)))

        low_energies, low_states = select_lowest(energies, states, num_states)

        final_energies = cp.concatenate((final_energies, low_energies))
        final_states = cp.concatenate((final_states, low_states))
        if i != 0:

            final_energies, final_states = select_lowest(final_energies,␣
 ↪final_states, num_states)

    return sort_by_key(final_energies, final_states, num_states)
```

[32]:
```
# test
import time
import os
import cupy as cp

from tqdm import tqdm
from IPython.utils.io import capture_output

n_range = list(range(15, 32))

# Kompilacja
Q = cp.random.uniform(-1, 1, size=(10, 10), dtype=cp.float32)
Q = cp.triu(Q)
with capture_output() as captured:
    brute_force_gpu(Q, 10)

times = []
for n in tqdm(n_range, desc="Wyczerpujące przeszukiwanie dla różnych n"):
    Q = cp.random.uniform(-1, 1, size=(n, n), dtype=cp.float32)
    Q = cp.triu(Q)
    start = time.time()
    with capture_output() as captured:
        brute_force_gpu(Q, num_states=10, sweep_size_exponent=min(n, 21),␣
 ↪threadsperblock=512)
    end = time.time()
    times.append(end - start)



plt.figure(figsize=(8, 5))
plt.plot(n_range, times, marker='o', linestyle='-', color='blue',␣
 ↪label='Execution Time')
plt.xlabel('Rozmiar (ilość spinów)')
plt.xticks(n_range)
plt.yscale('log', base=10)
plt.ylabel('Czas obliczeń (s)')
plt.grid(True)
plt.show()
```

```
Wyczerpujące przeszukiwanie dla różnych n: 100%|      | 17/17 [00:03<00:00,
4.56it/s]
```


Czas obliczeń (s)
10^0
10^{-1}
10^{-2}
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rozmiar (ilość spinów)

3