In [1]:
```
from hamiltonian import HamiltonianSmall, Hamiltonian
```

In [2]:
```
lih_small = HamiltonianSmall('LiH', 1.5) # this encoding uses reduction tec
beh2_small = HamiltonianSmall('BeH2', 1.3) # this encoding uses reduction t

# 4 qubits
h2_jw_4 = Hamiltonian('H2_STO3g_4qubits', 'jw')
h2_parity_4 = Hamiltonian('H2_STO3g_4qubits', 'parity')
h2_bk_4 = Hamiltonian('H2_STO3g_4qubits', 'bk')

# 8 qubits
h2_jw = Hamiltonian('H2_6-31G_8qubits', 'jw')
h2_parity = Hamiltonian('H2_6-31G_8qubits', 'parity')
h2_bk = Hamiltonian('H2_6-31G_8qubits', 'bk')

# 12 qubits
lih_jw = Hamiltonian('LiH_STO3g_12qubits', 'jw')
lih_parity = Hamiltonian('LiH_STO3g_12qubits', 'parity')
lih_bk = Hamiltonian('LiH_STO3g_12qubits', 'bk')

# 14 qubits
h2o_jw = Hamiltonian('H2O_STO3g_14qubits', 'jw')
h2o_parity = Hamiltonian('H2O_STO3g_14qubits', 'parity')
h2o_bk = Hamiltonian('H2O_STO3g_14qubits', 'bk')

beh2_jw = Hamiltonian('BeH2_STO3g_14qubits', 'jw')
beh2_parity = Hamiltonian('BeH2_STO3g_14qubits', 'parity')
beh2_bk = Hamiltonian('BeH2_STO3g_14qubits', 'bk')

# 16 qubits
nh3_jw = Hamiltonian('NH3_STO3g_16qubits', 'jw')
nh3_parity = Hamiltonian('NH3_STO3g_16qubits', 'parity')
nh3_bk = Hamiltonian('NH3_STO3g_16qubits', 'bk')

# 20 qubits
c2_jw = Hamiltonian('C2_STO3g_20qubits', 'jw')
c2_parity = Hamiltonian('C2_STO3g_20qubits', 'parity')
c2_bk = Hamiltonian('C2_STO3g_20qubits', 'bk')
```

In [3]:
```
hamiltonians = {"lih_small": lih_small,
                "beh2_small": beh2_small,
                "h2_jw_4": h2_jw_4,
                "h2_parity_4": h2_parity_4,
                "h2_bk_4": h2_bk_4,
                "h2_jw": h2_jw,
                "h2_parity": h2_parity,
                "h2_bk": h2_bk}
```

In [4]:
```
%time h2_jw_4.pauli_rep.ground()
%time h2_jw.pauli_rep.ground()
%time lih_jw.pauli_rep.ground()
%time energy, state = h2o_jw.pauli_rep.ground()
# %time energy, state = nh3_jw.pauli_rep.ground() # takes about 4 minutes
```

```
CPU times: user 22.5 ms, sys: 4.19 ms, total: 26.7 ms
Wall time: 23.2 ms
CPU times: user 501 ms, sys: 26.2 ms, total: 528 ms
Wall time: 418 ms
CPU times: user 4.32 s, sys: 431 ms, total: 4.76 s
Wall time: 2.61 s
CPU times: user 22.3 s, sys: 2.39 s, total: 24.7 s
Wall time: 10.8 s
```

In [5]:
```
%time h2_jw_4.pauli_rep.ground(multithread=True)
%time h2_jw.pauli_rep.ground(multithread=True)
%time lih_jw.pauli_rep.ground(multithread=True)
%time energy, state = h2o_jw.pauli_rep.ground(multithread=True)
# %time energy, state = ammonia_jw.pauli_rep.ground(multithread=True) # tak
```

```
CPU times: user 49.7 ms, sys: 50.7 ms, total: 100 ms
Wall time: 318 ms
CPU times: user 182 ms, sys: 79.8 ms, total: 262 ms
Wall time: 462 ms
CPU times: user 1.42 s, sys: 254 ms, total: 1.67 s
Wall time: 949 ms
CPU times: user 22.7 s, sys: 3.7 s, total: 26.4 s
Wall time: 4.74 s
```

Variance formula

$$\Var[\nu] = \sum_{\Qarrow,\Rarrow} f_\beta(\Qarrow,\Rarrow)\alpha_{\Qarrow}\alpha_{\Rarrow}\tr(\rho\Qarrow\Rarrow) - \tr(\rho H_0)^2$$

In [6]:
```
ham = h2_jw_4
energy, state = ham.pauli_rep.ground()
β = ham.pauli_rep.local_dists_uniform()
%time ham.pauli_rep.variance_local(energy, state, β)
```

```
CPU times: user 22.9 ms, sys: 2.99 ms, total: 25.9 ms
Wall time: 3.25 ms
```

Out[6]:
```
1.9710775636478903
```

In [7]:
```
print("ell_1: ", ham.pauli_rep.variance_ell_1(energy))
```

```
ell_1:  2.493466775932121
```

In [8]:
```
# this code is from an old idea, which ultimately did not work well.

β = ham.pauli_rep.local_dists_pnorm(1)
print("1_norm: ", ham.pauli_rep.variance_local(energy, state, β))

β = ham.pauli_rep.local_dists_pnorm(2)
print("2_norm: ", ham.pauli_rep.variance_local(energy, state, β))

β = ham.pauli_rep.local_dists_pnorm('infinity')
print("max_norm: ", ham.pauli_rep.variance_local(energy, state, β))
```

```
1_norm:  60.38753187541104
2_norm:  22.36573743732807
max_norm:  13.643445555482327
```

Variance optimisation (method=diagonal)

This is not the correct optimisation. However it

- gives good results;
- is quicker than the full optimisation problem;
- is convex (so local minimums are global);
- does not need access to the Hartree-Fock bitstring for the encoding.

Diagonal minimisation asks us to find $\{\beta_{i,P}\}$ in order to minimise:

$$\sum_{\text{\Qarrow}} \alpha^2_{\text{\Qarrow}} \prod_{i \in \text{\supp(\Qarrow)}} \beta^{-1}_{i,Q_i} \qquad \text{subject to} \qquad \beta_{i,X} + \beta_{i,Y} + \beta_{i,Z} = 1 \; \forall i, \qquad \beta_{i,P} \geq 0$$

And we have an implementation using Lagrange multipliers

In [9]:
```
β = ham.pauli_rep.local_dists_optimal('diagonal', 'scipy')
print(ham.pauli_rep.variance_local(energy, state, β))
```

```
1.8555808969284264
```

In [10]:
```
β = ham.pauli_rep.local_dists_optimal('diagonal', 'lagrange')
print(ham.pauli_rep.variance_local(energy, state, β))
```

```
1.855583049884943
```

In [11]:
```
# time comparison lih_jw has 12 qubits

ham = lih_jw
%time β_scipy = ham.pauli_rep.local_dists_optimal('diagonal', 'scipy')
%time β_lagrange = ham.pauli_rep.local_dists_optimal('diagonal', 'lagrange'

energy, state = ham.pauli_rep.ground(multithread=True)
var_scipy = ham.pauli_rep.variance_local(energy, state, β_scipy)
var_lagrange = ham.pauli_rep.variance_local(energy, state, β_lagrange)

discrepancy = abs(var_scipy - var_lagrange)
print("discrepancy between variances: ", discrepancy)
```

```
CPU times: user 1min 9s, sys: 9.1 s, total: 1min 18s
Wall time: 13.2 s
CPU times: user 2.35 s, sys: 83.1 ms, total: 2.43 s
Wall time: 1.86 s
discrepancy between variances:  4.931720457079791e-05
```

Variance optimisation (method=mixed)

This is the full optimisation problem. It requires access to the Hartree-Fock bitstring m or `bitstring_HF` so that the HF state reads $\frac{1}{2^n}\bigotimes_{i=1}^{n}(I+m_iZ)$

In the JW encoding these are:

- H2 = `1010` (on four qubits)
- H2 = `10001000`
- LiH = `100000100000`
- H2O = `11111001111100`
- BeH2 = `11100001110000`
- NH3 = `1111100011111000` You can retrieve them by calling `Hamiltonian.read_bitstring_HF()`

Consider the set of influential pairs:

$$\mathcal{I}_{\text{comp}} = \{(\text{\Qarrow}, \text{\Rarrow}) \mid \text{for all } i, \text{either } Q_i = R_i, \text{ or } \{Q_i, R_i\} = \{I, Z\}\}$$

Then the cost function to optimise will be:

$$\text{cost}(\{\beta_i\}_{i=1}^{n}) = \sum_{\text{\Qarrow},\text{\Rarrow}\in\mathcal{I}_{\text{comp}}} \alpha_{\text{\Qarrow}}\alpha_{\text{\Rarrow}} \prod_{i|Q_i=R_i\neq I} \beta_{i,Q_i}^{-1} \prod_{i|Q_i\neq R_i} m_i$$

Warning, the small Hamiltonians don't follow the pattern because they use other reduction techniques

In [12]:
```
ham = h2_jw
bitstring_HF = ham.read_bitstring_HF()
β = ham.pauli_rep.local_dists_optimal('mixed', 'scipy', bitstring_HF=bitstr
print(ham.pauli_rep.variance_local(energy, state, β))
```

```
/opt/anaconda3/envs/qiskit18/lib/python3.8/site-packages/scipy/optimize/_
hessian_update_strategy.py:183: UserWarning: delta_grad == 0.0. Check if
the approximated function is linear. If the function is linear better res
ults can be obtained by defining the Hessian as zero instead of using qua
si-Newton approximations.
  warn('delta_grad == 0.0. Check if the approximated '
```

```
-108.82360846362083
```

Variance after LDF grouping

We should use 1-norm sampling for κ

$$\Var[\nu] = \left(\sum_{k=1}^{n_c} \frac{1}{\kappa_k} \sum_{\Qarrow,\Rarrow \in C^{(k)}} \alpha_{\Qarrow} \alpha_{\Rarrow} \prod_{i \in \supp(\Qarrow\Rarrow)} \langle \Qarrow\Rarrow \rangle \right) - \langle$$

In [13]:
```
from var import variance_ldf, kappa_1norm #kappa_uniform
```

In [14]:
```
ldf = ham.ldf()
kappa = kappa_1norm(ldf)
energy_tf = ham.pauli_rep.energy_tf(energy)
variance_ldf(ldf, state, kappa, energy_tf)
```

Out[14]:
```
-108.81078810508258
```

Benchmarking

In [15]:
```
def variances_dict(ham, β_diag=None, β_mix=None):
    pr = ham.pauli_rep
    dic = {}

    energy, state = pr.ground(multithread=True)
    print("energy :", energy)

    # ell_1
    var = pr.variance_ell_1(energy)
    print("ell 1: ", var)
    dic['ell_1'] = var

    # LDF with 1-norm sampling
    ldf = ham.ldf()
    kappa = kappa_1norm(ldf)
    energy_tf = pr.energy_tf(energy)
    var = variance_ldf(ldf, state, kappa, energy_tf)
    print("ldf 1norm: ", var)
    dic['ldf_1norm'] = var

    # uniform
    β_uniform = pr.local_dists_uniform()
    var = pr.variance_local(energy, state, β_uniform, multithread=True)
    print("uniform: ", var)
    dic['uniform'] = var

    # optimal (diagonal)
    if β_diag is not None:
        var = pr.variance_local(energy, state, β_diag, multithread=True)
        print("optimal diagonal: ", var)
        dic['optimal_diag'] = var

    # optimal (mixed)
    if β_mix is not None:
        var = pr.variance_local(energy, state, β_mix, multithread=True)
        print("optimal mixed: ", var)
        dic['optimal_mix'] = var

    return dic

from matplotlib import pyplot as plt

def variances_graph(variances):
    num_variances = len(variances)
    x = range(num_variances)
    height = list(variances.values())

    plt.bar(x, height)
    plt.xticks(x, list(variances.keys()), rotation=20)
    plt.title(title)

    plt.show()
```

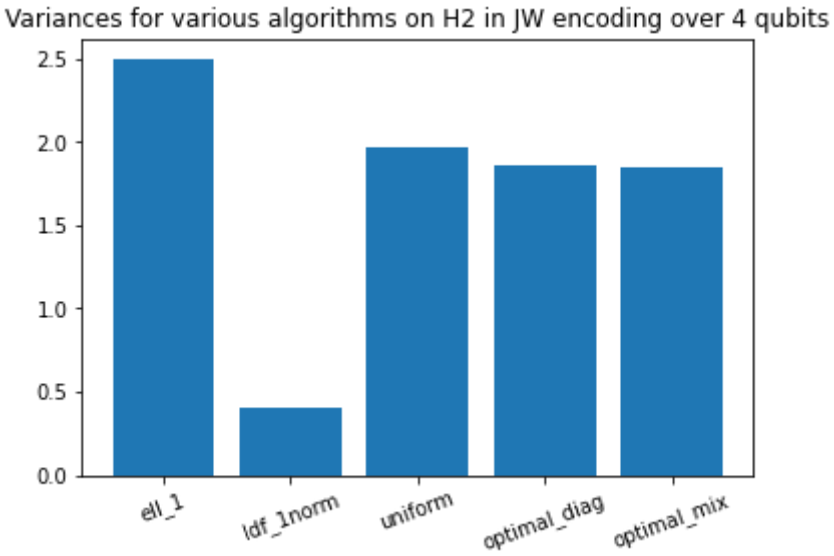
In [16]:
```
variances_ALL = {}
beta_optimal_ALL = {}
```

In [17]:
```
import time
```

In [18]:
```
def benchmarking(name, ham, title):
    bitstring_HF = ham.read_bitstring_HF()

    time_0 = time.time()
    β_diag = ham.pauli_rep.local_dists_optimal('diagonal', 'lagrange')
    time_1 = time.time()
    print("β_diag found. Time taken: ", time_1-time_0)
    β_mix = ham.pauli_rep.local_dists_optimal('mixed', 'scipy', bitstring_H
    time_2 = time.time()
    print("β_mix found.  Time taken: ", time_2-time_1)
    beta_optimal_ALL[name] = {'diagonal': β_diag, 'mixed': β_mix}

    variances_ALL[name] = variances_dict(ham, β_diag=β_diag, β_mix=β_mix)
    time_3 = time.time()
    print("Variances calculated.  Time taken: ", time_3-time_2)

    print("=====")
    print(title)
    print("=====")
    print(variances_ALL[name])
    print("=====")

    variances_graph(variances_ALL[name])
```

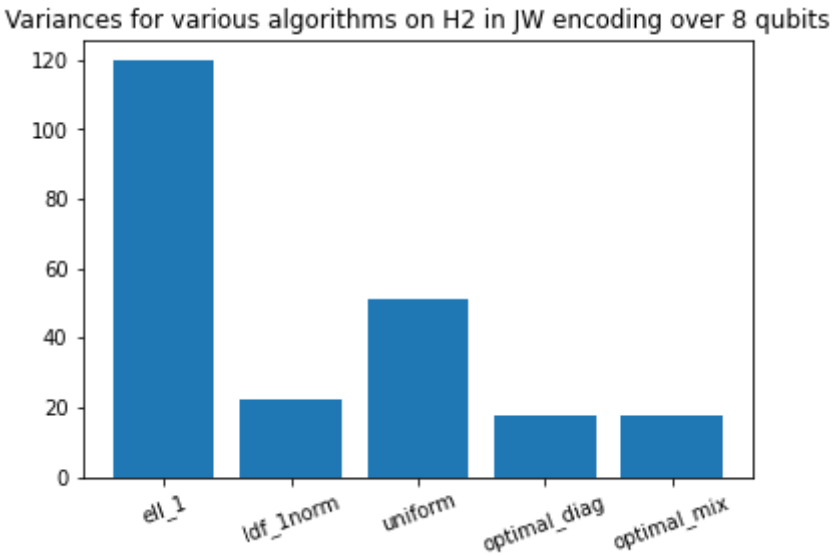
In [19]:
```
name = 'h2_jw_4'
ham = h2_jw_4
title = "Variances for various algorithms on H2 in JW encoding over 4 qubit

benchmarking(name, ham, title)
```

```
β_diag found. Time taken:  0.008041143417358398
β_mix found.  Time taken:  0.0512387752532959
energy : -1.857275030202382
ell 1:  2.4934667759321205
ldf 1norm:  0.40181970806913103
uniform:  1.9710775636478912
optimal diagonal:  1.8555830498849448
optimal mixed:  1.85465624251245
Variances calculated.  Time taken:  1.537532091140747
=====
Variances for various algorithms on H2 in JW encoding over 4 qubits
=====
{'ell_1': 2.4934667759321205, 'ldf_1norm': 0.40181970806913103, 'unifor
m': 1.9710775636478912, 'optimal_diag': 1.8555830498849448, 'optimal_mi
x': 1.85465624251245}
=====
```



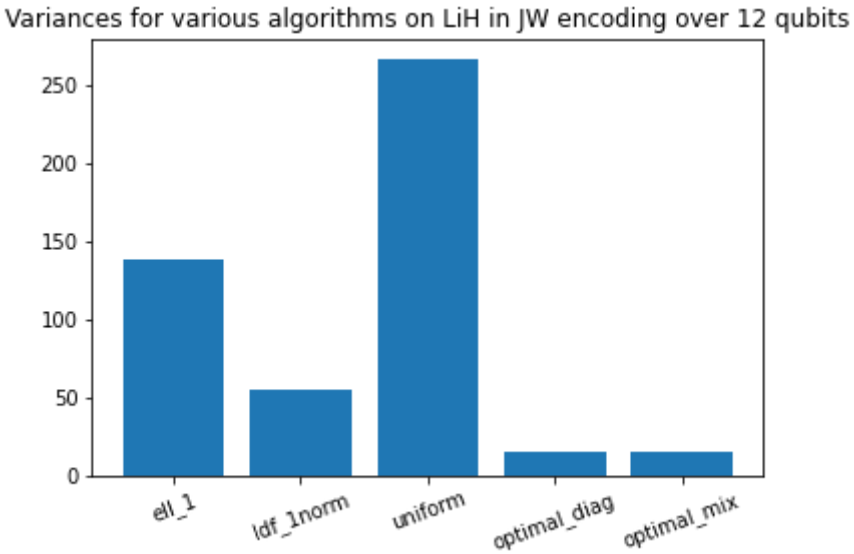
In [20]:
```
name = 'h2_jw'
ham = h2_jw
title = "Variances for various algorithms on H2 in JW encoding over 8 qubit

benchmarking(name, ham, title)
```

```
β_diag found. Time taken:  0.29004907608032227

/opt/anaconda3/envs/qiskit18/lib/python3.8/site-packages/scipy/optimize/_
hessian_update_strategy.py:183: UserWarning: delta_grad == 0.0. Check if
the approximated function is linear. If the function is linear better res
ults can be obtained by defining the Hessian as zero instead of using qua
si-Newton approximations.
  warn('delta_grad == 0.0. Check if the approximated '

β_mix found.  Time taken:  19.799096822738647
energy : -1.860860555520756
ell 1:  119.67906001905912
ldf 1norm:  22.259477922991145
uniform:  51.399820213875834
optimal diagonal:  17.741947811137166
optimal mixed:  17.457190704104455
Variances calculated.  Time taken:  2.2279322147369385
=====
Variances for various algorithms on H2 in JW encoding over 8 qubits
=====
{'ell_1': 119.67906001905912, 'ldf_1norm': 22.259477922991145, 'uniform':
51.399820213875834, 'optimal_diag': 17.741947811137166, 'optimal_mix': 1
7.457190704104455}
=====
```



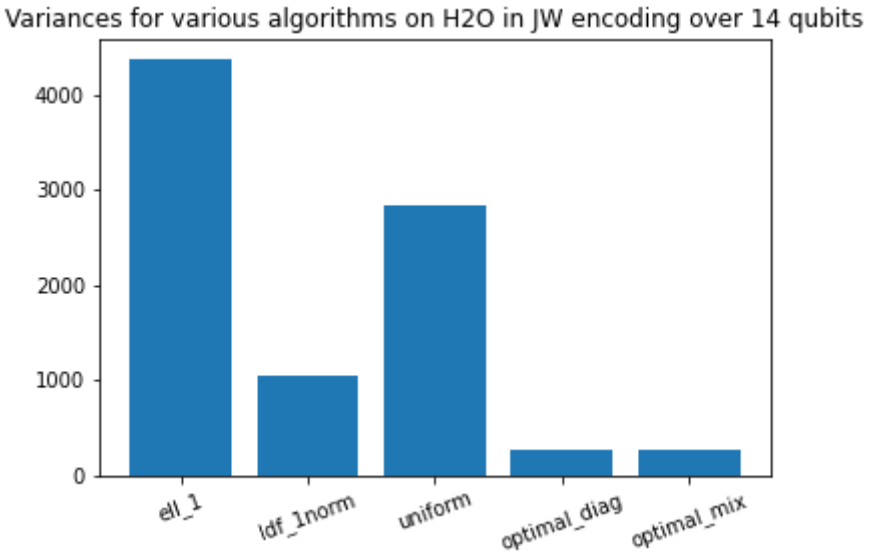
In [21]:
```
name = 'lih_jw'
ham = lih_jw
title = "Variances for various algorithms on LiH in JW encoding over 12 qub

benchmarking(name, ham, title)
```

```
β_diag found. Time taken:  1.984928846359253
β_mix found.  Time taken:  248.5241379737854
energy : -8.908299431473665
ell 1:  138.38018090986623
ldf 1norm:  54.15086386372376
uniform:  265.6353233020801
optimal diagonal:  14.792751908498898
optimal mixed:  14.87757644953169
Variances calculated.  Time taken:  17.072442054748535
=====
Variances for various algorithms on LiH in JW encoding over 12 qubits
=====
{'ell_1': 138.38018090986623, 'ldf_1norm': 54.15086386372376, 'uniform':
265.6353233020801, 'optimal_diag': 14.792751908498898, 'optimal_mix': 14.
87757644953169}
=====
```



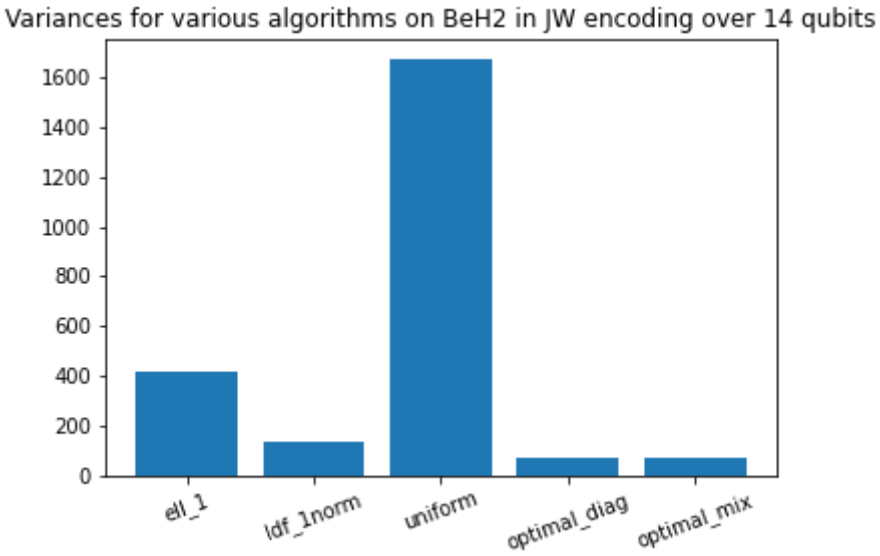
In [22]:
```
name = 'h2o_jw'
ham = h2o_jw
title = "Variances for various algorithms on H2O in JW encoding over 14 qub

benchmarking(name, ham, title)
```

```
β_diag found. Time taken:  4.18767786026001
β_mix found.  Time taken:  1064.8798830509186
energy : -83.59943020533808
ell 1:  4363.497773126062
ldf 1norm:  1041.823501346878
uniform:  2839.03946821896
optimal diagonal:  257.5439313165558
optimal mixed:  254.14108178948788
Variances calculated.  Time taken:  152.60770511627197
=====
Variances for various algorithms on H2O in JW encoding over 14 qubits
=====
{'ell_1': 4363.497773126062, 'ldf_1norm': 1041.823501346878, 'uniform': 2
839.03946821896, 'optimal_diag': 257.5439313165558, 'optimal_mix': 254.14
108178948788}
=====
```



In [24]:
```
name = 'beh2_jw'
ham = beh2_jw
title = "Variances for various algorithms on BeH2 in JW encoding over 14 qu

benchmarking(name, ham, title)
```

```
β_diag found. Time taken:  2.3394269943237305
β_mix found.  Time taken:  644.2576398849487
energy : -19.045049602808028
ell 1:  418.2697172297552
ldf 1norm:  135.4290971990617
uniform:  1670.0146708893778
optimal diagonal:  67.59976246292501
optimal mixed:  67.48962329524991
Variances calculated.  Time taken:  66.65773487091064
=====
Variances for various algorithms on BeH2 in JW encoding over 14 qubits
=====
{'ell_1': 418.2697172297552, 'ldf_1norm': 135.4290971990617, 'uniform': 1
670.0146708893778, 'optimal_diag': 67.59976246292501, 'optimal_mix': 67.4
8962329524991}
=====
```



In [23]:
```
# mac book pro is too weak for this!

#name = 'nh3_jw'
#ham = nh3_jw
#title = "Variances for various algorithms on NH3 in JW encoding over 16 qu
```

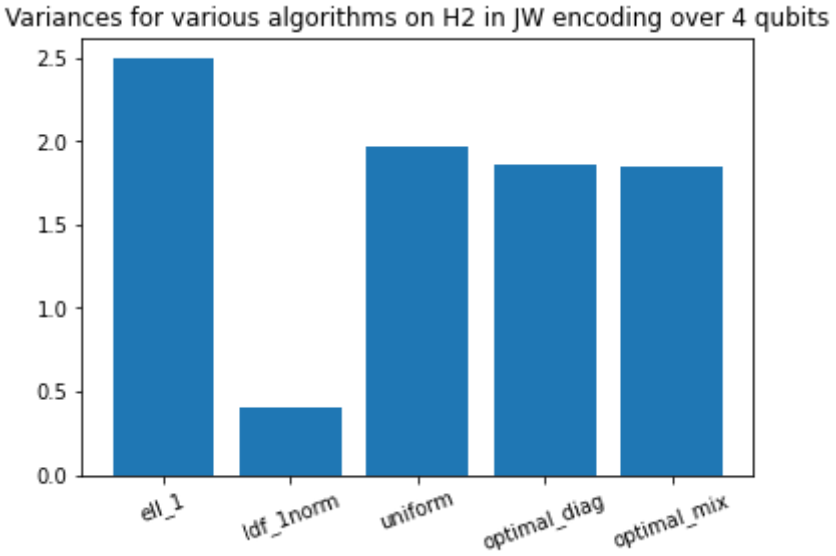
In [ ]:

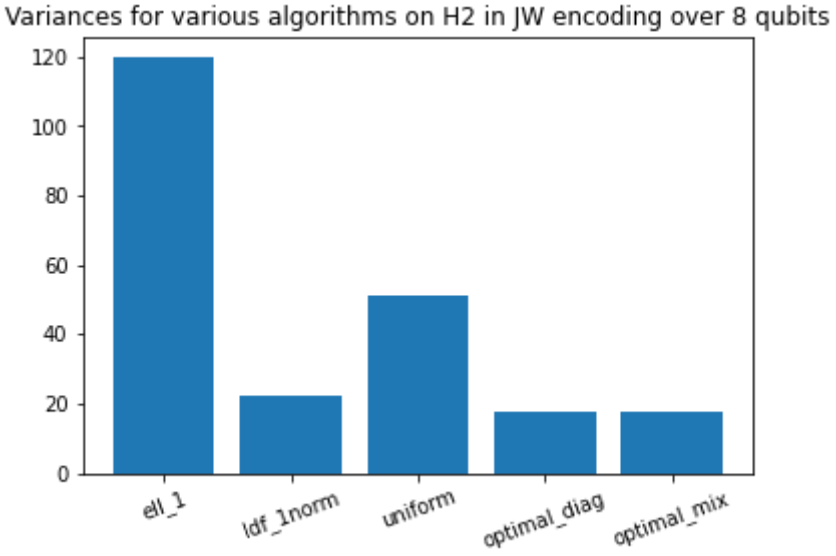
In [ ]:

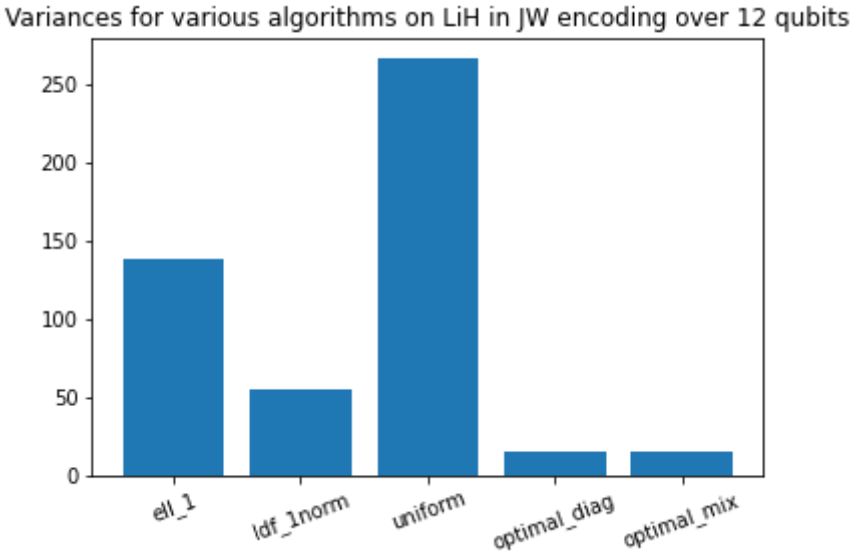
In [29]:
```
def save_variances_graph(variances, title, name):
    num_variances = len(variances)
    x = range(num_variances)
    height = list(variances.values())

    plt.bar(x, height)
    plt.xticks(x, list(variances.keys()), rotation=20)
    plt.title(title)

    file = '../images/5algos_jw_encoding/{}'.format(name)
    plt.savefig(file, dpi=300)
    plt.show()
```

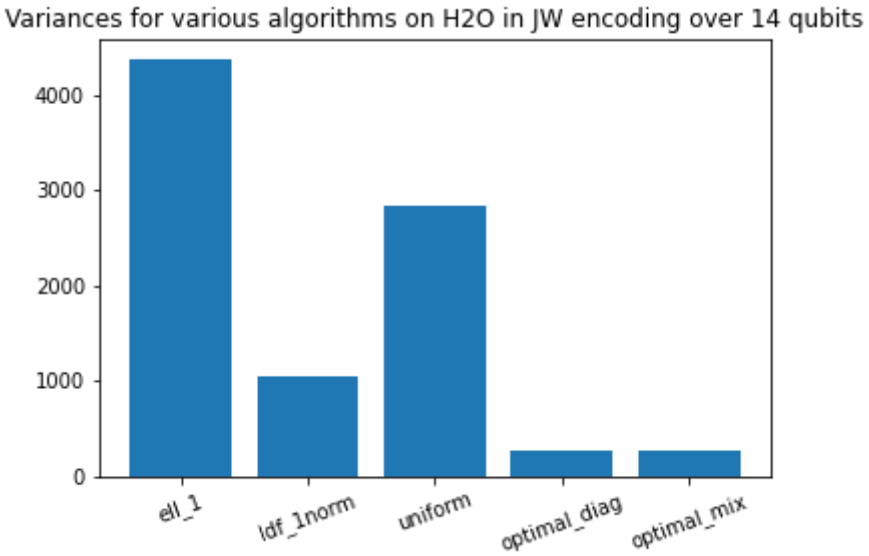
In [30]:
```
graph_data = {"h2_jw_4": [variances_ALL["h2_jw_4"],
                          "Variances for various algorithms on H2 in JW enc
                          "h2_jw_4q_1min"],
              "h2_jw": [variances_ALL["h2_jw"],
                        "Variances for various algorithms on H2 in JW encod
                        "h2_jw_8q_1min"],
              "lih_jw": [variances_ALL["lih_jw"],
                         "Variances for various algorithms on LiH in JW enc
                         "lih_jw_12q_5min"],
              "h2o_jw": [variances_ALL["h2o_jw"],
                         "Variances for various algorithms on H2O in JW enc
                         "h2o_jw_14q_20min"],
              "beh2_jw": [variances_ALL["beh2_jw"],
                          "Variances for various algorithms on BeH2 in JW e
                          "beh2_jw_14q_14min"]}
```

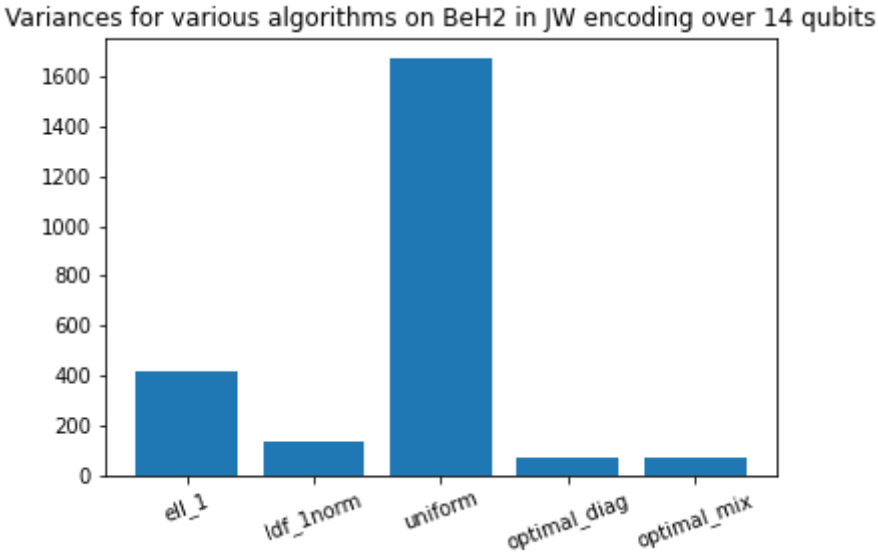
In [31]:
```
#for data in graph_data.values():
#    save_variances_graph(*data)
```








Variances for various algorithms on H2O in JW encoding over 14 qubits
4000
3000
2000
1000
0
ell_1
ldf_1norm
uniform
optimal_diag
optimal_mix


Variances for various algorithms on BeH2 in JW encoding over 14 qubits
1600
1400
1200
1000
800
600
400
200
0
ell_1
ldf_1norm
uniform
optimal_diag
optimal_mix

In [ ]:

In [ ]: