Opleiding Informatica

Ground State approximation with Product States
using Graph Theory

Christian Martens

Supervisors:
Evert van Nieuwenburg, Patrick Emonts & Jordi Tura

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

27/06/2023

Abstract

Ground states of one-dimensional quantum systems can be approximated efficiently (i.e. in polynomial time) as product states. Finding the ground state efficiently can be mapped to a graph problem in which we find the k shortest paths. We present two methods to improve the run time of finding the k shortest paths for this specific application. In this application the problem instances are comprised of directed acyclic graphs (DAGs). The first method aims to improve the run time by slightly modifying the graph whilst maintaining the ability to perform general graph algorithms on it, with an expected quadratic speedup. The second method is the implementation of Eppstein's algorithm, which has a better theoretical time complexity than the algorithms that have been used so far.

Contents

1 Introduction **1**

2 Preliminaries **4**
2.1 Notation for notions from linear algebra . . . 4
2.2 Quantum Mechanics . . . 5
2.2.1 Systems and states . . . 5
2.2.2 Energy and the Hamiltonian . . . 6
2.2.3 Ising model . . . 7
2.3 Graph Theory . . . 7

3 Methods **9**
3.1 Graph Augmentation . . . 9
3.1.1 Time complexity . . . 10
3.2 Eppstein's algorithm . . . 10
3.2.1 Shortest path tree . . . 11
3.2.2 Sidetrack costs . . . 12
3.2.3 Sidetrack edge heaps for v . . . 13
3.2.4 Graph creation . . . 15
3.2.5 Graph modification . . . 15
3.2.6 Heap creation . . . 16
3.2.7 Obtaining the paths from the heap . . . 17

4 Results **18**
4.1 Graph Augmentation . . . 18
4.2 Eppstein's algorithm . . . 19
4.2.1 Randomly weighted graphs . . . 19
4.2.2 Slightly perturbed graphs . . . 20
4.2.3 Eppstein analysis . . . 20

5 Conclusion **21**

References **23**

A Computer Hardware **23**
A.1 Computer A . . . 23
A.2 Computer B . . . 23

1 Introduction

Generally, finding the ground state of quantum systems and the corresponding energy is NP-hard as shown by Kitaev, who draws a parallel between this problem and the well known *constraint satisfaction problem* (CSP) [GKSV03]. In this CSP a set of constraints is given with regard to some set of N classical particles that can be in q states. The decision problem is now to determine if more than a given number of constraints can be satisfied. Even for one-dimensional systems this problem remains hard [AGIK09].
Despite the general hardness of this problem, Schuch and Cirac show that the mean-field solutions that neglect correlations between the particles can be computed in polynomial time for one-dimensional (1D) systems [SC10]. They show this by first explaining how to efficiently solve for the minimal energy of a classical spin chain and then showing how in a similar way the ground state energy of 1D quantum systems can be approximated efficiently.
A classical spin chain is a 1D chain of spin systems where each of the sites has a spin that can have any of d values. The minimal energy can be solved sequentially by going over the chain from left to right. At every step, the minimal energy of the left half of the chain will only depend on the value of the spin at the current position. The spins that will still influence the optimal energy of the part of the chain to the right of the current position are said to be in the boundary setting. In the case of the classical system only the spin at the left of the current site is in the boundary setting. For each site, the energy needs to be minimized over d settings of the site to the left of it and as the current site can also be in d settings the computational cost for a chain of N sites will thus be Nd^2. This algorithm not only yields the optimal energy but also the corresponding ground state.
The mean-field solution of a 1D quantum system with Hamiltonians of the form

$$H = \sum_{k=1}^{N-1} H_{k,k+1}, ||H_{k,k+1}||_\infty \leq 1 \tag{1}$$

can be found in a similar way. The mean-field approach considers the quantum system in the form of a product state, which is the tensor product of N smaller systems: $|\psi\rangle = \otimes^N |\psi\rangle, |\psi\rangle \in \mathbb{C}^d$. This is an incomplete description of the system as it only accounts for local interactions, which means we cannot find the global optimum with this description. However, the global optimum can be well approximated for systems that mostly have local interactions. The minimization problem with respect to the product state description is then as follows (where all states are normalized):

$$E = \min_{|\psi_1\rangle,\ldots,|\psi_N\rangle} \sum_{k=1}^{N-1} \langle \psi_k, \psi_{k+1} | H_{k,k+1} | \psi_k, \psi_{k+1} \rangle \tag{2}$$

Formula 2 shows the minimization of the sum of the energies of all subsequent pairs of neighboring sites. $|\psi_k, \psi_{k+1}\rangle$ denotes a local subsystem in the chain in which we consider sites k and $k+1$. The local Hamiltonian corresponding to that subsystem is denoted by $H_{k,k+1}$. Because the correlations are neglected in the mean field approach, the minimal energy approximation of the system only depends on the energy of the local subsystems. As so, the solution can be obtained using a dynamic programming approach by dividing the problem into local sub problems and combining the results. Because the parameters $|\psi_k\rangle$ are continuous and the Hamiltonian that we use to compute the energy is of exponential size, we cannot compute the energy exactly. Therefore, the minimization is

restricted to a discrete set of solutions for each site, called an ϵ-net. The discrete solutions in the ϵ-net are a set of vectors $|\psi_k^\alpha\rangle$ with $\alpha = 1, \dots \mathcal{A}$ such that

$$\forall |\psi_k\rangle \in \mathbb{C}^d \; \exists \alpha : \left\| \, |\psi_k\rangle\langle\psi_k| - |\psi_k^\alpha\rangle\langle\psi_k^\alpha| \, \right\|_1 \leq \epsilon \tag{3}$$

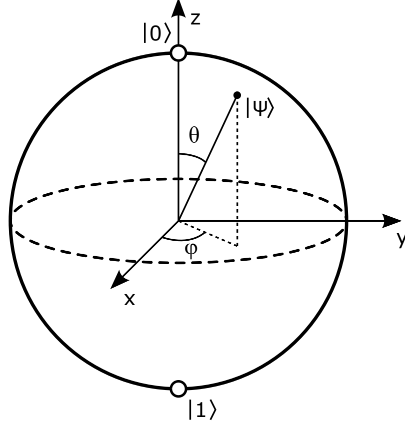
This formula describes how there exist vectors $|\psi_k^\alpha\rangle$ such that the norm of the difference of the projection matrices of $|\psi_k\rangle$ and $|\psi_k^\alpha\rangle$ is less than some value ϵ. The result of this for vectors in $\mathbb{C}^2$ can be visualised on the Bloch sphere as a net of points that are some bounded distance away from each other on the sphere, see Figure 2.

This reduces the algorithm to the same algorithm as for the 1D classical spin chain, where the number of discrete solutions that we consider is the number of settings each site can be in. This way the algorithm will yield the optimal solution in the set of all product states that can be made using the states that we consider on each site, scaling linearly in the number of particles and quadratically in the number of states per particle [AAI10]. The key feature of the ϵ-net is that because all solutions on a site differ by a bounded amount there is an upper bound on the error of the ground state energy as product state.

The work of this thesis is part of a larger ongoing project at Leiden University by Jordi Tura and Patrick Emonts. Their idea is that for the same type of Hamiltonians as in Formula 1, the structure of the Hamiltonian can be used to find the ground state approximation more efficiently. Building on the earlier works of Aharonov et al. and Schuch and Cirac, they are working on a method in which the ϵ-net is refined iteratively. The method works by finding the k lowest energy states of the system and increasing the resolution of the net in the area where the k lowest energy states were found. This is done by disregarding those solutions on the net that were not part of the k lowest energies and adding a new discrete set of possible solutions in-between the remaining solutions. In the next iteration we consider the solutions that were not disregarded, together with the solutions from the new set and find the k lowest energy states over the set of product states that use this united set of solutions. This process is then repeated until it converges to the minimum value. Note that because solutions are disregarded, the net is not an ϵ-net anymore as the constraint of formula 3 does not hold anymore. However, according to their idea the ground state can still be approximated using this method due to the structure of these specific Hamiltonians. The efficiency comes from the fact that a lower number of solutions is now needed to achieve the same accuracy as in the previously described ϵ-net method, or in other words a higher accuracy can be achieved by considering the same number of solutions.

To benchmark this method they make use of the 1D transverse field Ising model with vanishing transverse field (the field is set to 0). The reason to use this model is because in the case of the field being set to 0, the model can be reduced to the classical Ising model with field 0 of which the solution can easily be obtained. This means the model can be used to verify that the method works. Although it is known how to solve this problem using the classical Ising model, it is treated in the quantum mechanical way so that the same method can later be applied on quantum versions of the model where the transverse field is not 0.

The implementation of this extended method can be understood by looking at the minimization problem from a graph perspective. To find the minimal energy as given by Formula 2 we need to consider all possible combinations of states that neighbouring sites ψ_k and ψ_{k+1} of the product state can be in. To represent all states from the discrete set of states that form the ϵ-net, we create a set of vertices for each site, where each state of a site is represented by a vertex. To represent



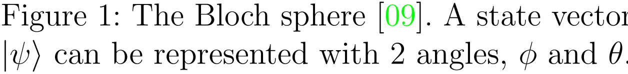


Figure 1: The Bloch sphere [09]. A state vector $|\psi\rangle$ can be represented with 2 angles, ϕ and θ.

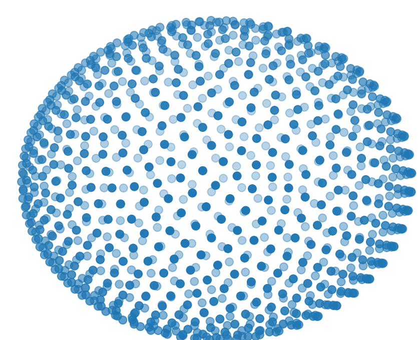


Figure 2: The ϵ-net construction visualised as the Bloch sphere covered in points that are a bounded distance away from each other.

all combinations of states of a local system of neighbouring sites we add directed edges from all vertices of site ψ_k to all vertices of site ψ_{k+1}. The sets of vertices are now topologically ordered from ψ_1 to ψ_N, where each set of vertices is only connected to the set of vertices of the succeeding site, see Figure 3. The energy that corresponds to a combination of states of two neighboring sites can be computed using the two-local Hamiltonian as $\langle \psi_k, \psi_{k+1} | H_{k,k+1} | \psi_k, \psi_{k+1} \rangle$, just like in the sum of Formula 2. This energy is added as weight on the edge that connects the two states. Finding the k lowest energy states on the set of product states of the complete system now corresponds to finding the k shortest paths from the vertex set of site ψ_1 to the one of site ψ_N in the graph.
In this thesis we show how the usage of these graphs in the new iterative method by J. Tura and P. Emonts can be optimized by adjusting the graphs and we explore how better time complexities can be achieved on finding the desired properties of the graphs that correspond to the ground state approximation and its energy. As we make use of a layered graph structure as in Figure 3, we may refer to the vertex sets $\psi_1 \ldots \psi_N$ as layers. The states represented by the vertices in each set may also be referred to as points.

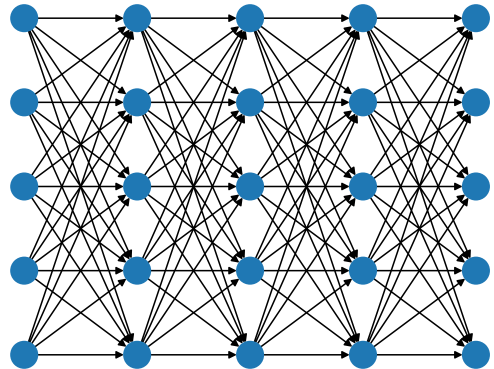


Figure 3: Example graph demonstrating the structure of the graphs that are used to find the minimal energy. This graph has 5 "columns" of vertices representing 5 sites from left to right and 5 vertices in each column vertically representing the states a site can be in. Generally the graphs can have any number of sites and states per site.

2 Preliminaries

2.1 Notation for notions from linear algebra

In quantum mechanics we often make use of linear algebra in the vector space of $\mathbb{C}^n$. We usually denote vectors in this vector space and other notions using a special notation known as Dirac notation. In Dirac notation a vector is denoted as

$$|\psi\rangle. \tag{4}$$

Here ψ is a label for the vector and $|\cdot\rangle$ indicates that the object is a vector. Physicists also often use special labels for some vectors, in particular we will use the following convention:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5}$$

In Table 1 we summarize some of the important notations that are used [NC10, p. 62]. The tensor product of vectors $|\phi\rangle$ and $|\psi\rangle$ can also be abbreviated as $|\phi\rangle|\psi\rangle$ or even $|\phi, \psi\rangle$.
In quantum mechanics we are also often interested in matrices that have certain properties, in particular *Hermitian* matrices and *unitary* matrices.
The Hermitian matrix is a matrix that is equal to its own conjugate transpose, that is

$$H = H^\dagger. \tag{6}$$

$H^\dagger$ is also referred to as the adjoint of the H matrix. These matrices have the property that their eigenvalues are always real-valued.
A Unitary matrix is a matrix of which the conjugate transpose is equal to its own inverse, that is

$$U^\dagger = U^{-1}, \tag{7}$$

Notation	Description
$\|\psi\rangle$	Vector. Also known as a *ket*
$\langle\psi\|$	The conjugate transpose of the vector $\|\psi\rangle$, also known as a *bra*
$\langle\phi\|\psi\rangle$	Inner product between the vectors $\|\phi\rangle$ and $\|\psi\rangle$
$\|\phi\rangle\langle\psi\|$	Outer product between the vectors $\|\phi\rangle$ and $\|\psi\rangle$
$\|\phi\rangle \otimes \|\psi\rangle$	Tensor product of $\|\phi\rangle$ and $\|\psi\rangle$
$\langle\phi\|A\|\psi\rangle$	Inner product between $\|\phi\rangle$ and the matrix vector product $A\|\psi\rangle$
$A^\dagger$	Conjugate transpose of the matrix A

Table 1: Notations used for notions from linear algebra known as *Dirac* notation. It is sometimes also referred to as bra-ket notation.

which means that the product of the matrix with its conjugate transpose is equal to the identity matrix: $UU^\dagger = I$. An important property of the unitary matrices is that the norm of vectors is preserved under matrix vector multiplication.
One set of extremely useful matrices that are Hermitian and unitary are the Pauli matrices which are listed in Figure 4, they are also equal to their own inverse (involutory).

$$I \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$
$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Figure 4: The Pauli matrices.

With these notations we can explain the concepts from quantum mechanics to understand the motivation of this thesis.

2.2 Quantum Mechanics

2.2.1 Systems and states

In physics we usually talk about physical systems and states. Physical systems are a part of the physical universe that we try to describe and analyse [Bel12]. The state of a system is a set of variables that describes the system at a certain point in time [Mes66].
The first postulate of quantum mechanics tell us that we make use of a complex Hilbert space to describe the states of quantum systems, this is also known as the *state space*. The state a quantum system can be in is described using a unit vector in the system's state space, also known as the *state vector*.
Some physical systems can be described using two-dimensional state vectors, also known as qubits. Just like regular bits can be in the states 0 and 1, qubits can also be in two states represented by $|0\rangle$ and $|1\rangle$. The difference between bits and qubits is that qubits can also be in states other than these, namely, they can be in states that form a linear combination of $|0\rangle$ and $|1\rangle$ as show below:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{8}$$

These are also known as *superpositions* [NC10, p. 13]. $|0\rangle$ and $|1\rangle$ form an orthonormal basis for the state space, so by multiplying these vectors with the complex coefficients α and β we can form any state vector. We can view $|\alpha|^2$ and $|\beta|^2$ as the probabilities to measure the 0-state and 1-state respectively. As the probability to measure something must be 1, the sum of the probabilities to measure either of the states must also be 1, i.e. $|\alpha|^2 + |\beta|^2 = 1$ [Wil10]. As a result of this the state vector is a unit vector.

To make it easier to think about the state vectors it is common to visualise them. For high-dimensional vectors this becomes difficult but two-dimensional vectors can be represented on the *Bloch sphere*, see Figure 1. It may not be apparent how this can be done as one might think that four dimensions are needed to represent a two-dimensional complex vector, one dimension for the real and complex part of both dimensions. However, because of the restriction that the length of the vector must be one, the visualisation can be brought back to 3 dimensions [Blo46]. Two-dimensional state vectors can therefore also be represented by two angles, one for the polar and one for the azimuthal angle, and thus by 2 parameters. A result of this representation is that orthogonal vectors will be opposite to each other, as can be seen in Figure 1 where the vectors $|0\rangle$ and $|1\rangle$ are on the north and south pole of the sphere.

2.2.2 Energy and the Hamiltonian

With each state that a system can be in, we usually associate some energy. We are often interested in the state of a system that has the lowest energy, this state is also known as the *ground state*. A system's energy can be described using the *Hamiltonian*. Generally, the Hamiltonian can be seen as an energy function, taking a configuration of the system as an argument and yielding the corresponding energy. In quantum mechanics the Hamiltonian is an operator that corresponds to the total energy of a system and is a Hermitian matrix, this comes from the second postulate of quantum mechanics. Operator here means a function that maps vectors from one space to another. The different energy levels that the system can be in are described by the eigenvalues of the Hamiltonian. Note that because the Hamiltonian is Hermitian the eigenvalues will always be real-valued. The states that correspond to these energy levels are those state vectors that are the eigenvectors corresponding to these eigenvalues. Together they solve the eigenvalue equation

$$H|\psi\rangle = E|\psi\rangle \tag{9}$$

also known as the stationary Schrödinger equation, where E is an eigenvalue of H. Thus, if the state vector $|\psi\rangle$ is an eigenvector of H, the energy corresponding to this state can be computed by taking the inner product between $|\psi\rangle$ and $H|\psi\rangle$ and dividing by the norm of $|\psi\rangle$, which we denote as

$$E = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle}. \tag{10}$$

If our states are normalized, we can leave the division by the norm out of the equation and we can get the energy by just computing $\langle\psi|H|\psi\rangle$.

2.2.3 Ising model

The Ising model is a mathematical model for studying ferromagnetism in statistical mechanics [Gal99]. The 1D version of the model is concerned with a chain of sites, each to which a variable is assigned representing the spin of the site, which is an intrinsic form of angular momentum carried by elementary and composite particles [Mer98, Gri05].
In the classical Ising model, spins are represented by discrete variables and can be in either of the two states +1 or −1. For a configuration σ that describes a spin configuration $\sigma_k \in \{+1, -1\}$ for each site k, the Hamiltonian of this model can be denoted as

$$H(\sigma) = J \sum_{\langle ij \rangle} \sigma_i \sigma_j + \mu \sum_j h_j \sigma_j \tag{11}$$

The notation $\langle ij \rangle$ here denotes that sites i and j are nearest neighbors, so this formula tells us that each site has interactions with its nearest neighbors. The second term that is the sum over individual sites accounts for the effect of an external magnetic field.
In the transverse field Ising model, a quantum version of the model, spins are represented by qubits as the spin of a system can be in a superposition of states. The Hamiltonian for this model can be denotes as

$$H = J \sum_{\langle i,j \rangle} Z_i Z_j + h \sum_i X_i \tag{12}$$

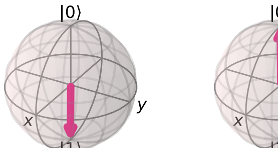
Here Z_k and X_k are the Pauli Z and X matrices, they represent the observable (a physical quantity that can be measured) corresponding to spin along the z and x axes. The first sum term again corresponds to the interactions between nearest neighbors and the second sum term to an external field. If the coefficient h that determines the strength of the field relative to the nearest neighbour interaction is 0, the model can be reduced to the classical model with field 0. First observe that we lose the second sum term when there is no transverse field:

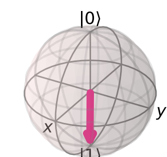
$$\text{for } h = 0: \ H = J \sum_{\langle i,j \rangle} Z_i Z_j \tag{13}$$

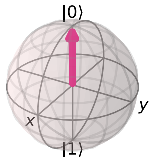
Now notice that we are only left with Z matrices. As these matrices have their eigenvalues on the diagonal, looking at the diagonal of the matrices suffices to compute the energy. We can now only assign one of the two eigenvalues from the diagonal to each Z matrix, which is equivalent to assigning one of the two configurations to each σ is in the classical model. We can now also easily reason about what the lowest energy state would be. The Z matrices have eigenvalues 1 and -1. As we are looking at neighbouring sites, their product with J must yield the lowest energy. If we know that J is positive, then the configuration of 1 for Z_i and -1 for Z_j or -1 for Z_i and 1 for Z_j will both yield $-J$ as the energy. This means that the ground states of this system are those states in which the spins are opposite to each other in an alternating fashion, see Figure 5.

2.3 Graph Theory

In order to compute the k shortest paths more efficiently we make use of several data structures. As we implement Eppstein's algorithm, many of the definitions that we use to explain the implementation







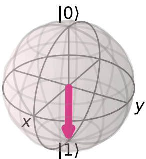


Figure 5: A one-dimensional spin systems in one of the ground states.

come naturally from his paper [Epp98].

Let us first define a *directed graph* or *digraph* $G = (V, E)$ to be a pair, where V is a set of vertices and E a set of ordered pairs of distinct vertices. Henceforth, n will denote the number of vertices $|V|$ and m the number of edges $|E|$. If (u, v) is a directed edge e from u to v, then $\text{tail}(e) = u$ and $\text{head}(e) = v$. The *weight* of an edge is denoted by $\ell(e)$. We define a path p to be a set of edges that connect two vertices. Note that the path from a vertex s to a vertex t will be a different set of edges than the path from t to s (if such a path exists), this is because an edge from vertex u to v cannot be used to go from v to u. The weight of a path $\ell(p)$ is then defined as the sum of the weights of the edges in the path. Let *distance* $d(s, t)$ denote the weight of the shortest path of all paths from vertex s to vertex t. Note that for the same reason the paths between s and t consist of different edge sets, $d(s, t)$ is not necessarily equal to $d(t, s)$ as the paths may consist of differently weighted edges or even a different number of edges.

To implement Eppstein's algorithm we use a lot of *heap*-like data structures. By *heap* we mean a binary tree where the children of each vertex are larger or equal to the parent, also known as a *min-heap*. A *D-heap* will denote a tree with the same weight-ordering property as the heap where each vertex has out-degree D. In balanced heaps, any set of values can be placed into the heap in linear time using the *heapify* operation and a new value can be inserted, or *pushed* onto the heap in logarithmic time [Epp98]. The operation of removing the smallest element from the heap or *popping* from the heap is also performed in logarithmic time, this is because the top element can be retrieved in $O(1)$, but if we remove the top element the heap needs to be restructured.

3 Methods

Previously, finding the k shortest paths between the first and last layer of the graph, corresponding to the k lowest energy states of the system, was implemented by iterating over all combinations of start and end nodes in the first and last layer respectively and performing either a variant of Dijkstra's algorithm or performing Waterman's [WB85] algorithm for each combination. The methods described in this thesis aim to improve upon this approach in two ways. The methods consist of changing the approach graph theoretically, by augmenting the graph, and an attempt to improve the run time by implementing an algorithm with a better time complexity, Eppstein's algorithm for the k shortest paths [Epp98].

3.1 Graph Augmentation

The run time of the calculations can be improved by augmenting the graph with two vertices. These vertices, which we will name s and t, will be added before and after the first and last layer respectively. Directed edges with weight 0 are added from s to all of the nodes in the first layer and from all the nodes in the last layer to t, see Figure 6. In the methods used on the original graph, different start and end node pairs were considered and the same algorithm was run for each pair. The result of this was that the information about paths in the graphs was lost in-between consecutive algorithm runs. The changes that we make to the graph have no physical interpretation. However, they allow us to keep the path information that previously was lost in-between consecutive algorithm runs applied to the original start and end node pairs. This allows us to speed up the run time without the need to modify existing k shortest path algorithms by applying a k shortest path algorithm with as starting vertex s and target vertex t.

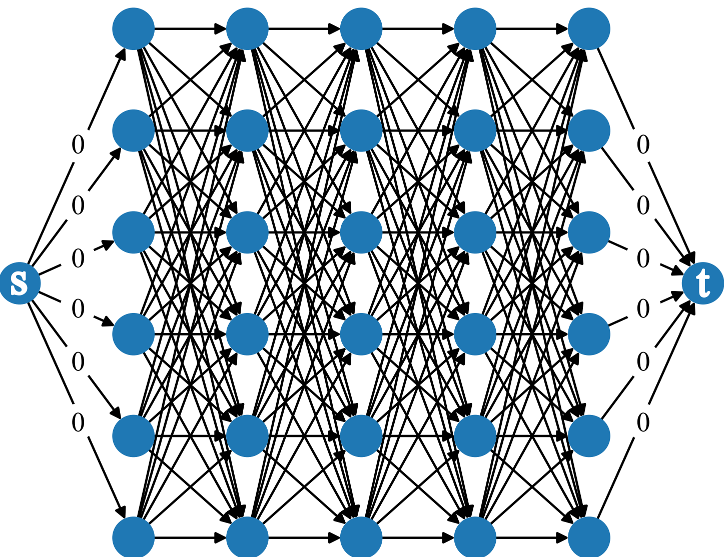


Figure 6: The modified example graph. Edges connected to verteces s and t have weight 0, the rest of the graph remains the same.

3.1.1 Time complexity

Assuming all sites have an equal number of q states to consider, (and thus an equal number of vertices for each layer in the graph), this method will provide a speedup of q^2. Let us see how this works when we consider the variant of Dijkstra's algorithm to find the k shortest paths which runs in $O(m + kn\log n)$ [Epp98].
If ℓ denotes the number of layers in the original unadjusted graph, then the number of vertices will be $n = \ell q$ and the number of edges in the graph will be $m = (\ell - 1)q^2$. The only change that is made when augmenting the graph is an addition of a constant number of 2 vertices and an addition of $2q$ edges. So after this modification we have $n = \ell q + 2$ vertices and $m = (\ell - 1)q^2 + 2q$ edges. Plugging in these values for n and m we find for the original graph

$$O(m + kn\log n) = \tag{14}$$
$$O((\ell - 1)q^2 + k(\ell q)\log(\ell q)) \tag{15}$$

and for the graph after modification

$$O(m + kn\log n) = \tag{16}$$
$$O(2q + (\ell - 1)q^2 + k(\ell q + 2)\log(\ell q + 2)) = \tag{17}$$
$$O((\ell - 1)q^2 + k(\ell q)\log(\ell q)) \tag{18}$$

Looking at the time complexity for the graph after modification in formula 17, we see that the 2's in the number of nodes are constants and their addition can be upper bound by ℓq and for the edges the term $2q$ scales less than $(\ell - 1)q^2$ due to the higher power in the second term and thus can be upper bound by $(\ell - 1)q^2$. Therefore, a run on the augmented graph has the same asymptotic time complexity as a single run on the original graph. However, for the original graph, the algorithm had to be run q^2 times whereas with the augmented graph only once. This is why the augmented graph method is quadratically faster than the original method.

3.2 Eppstein's algorithm

Another attempt at improving the run time of finding the k shortest paths lies in improving the time complexity by implementing a different algorithm. If m and n denote edges and vertices, then applying the variant of Dijkstra's algorithm to find the k shortest paths will take $O(m + kn\log n)$ [Epp98]. Waterman's algorithm which finds all paths in some range of $p\%$ above the minimal path weight has a complexity of $O(Rn + kR\sqrt{n})$, where R denotes the average number of edges emanating from each vertex and each path contains $\sqrt{n}$ vertices [WB85]. Eppstein's algorithm in general has an asymptotic time complexity of $O(m + n\log n + k)$. However, in DAG's $O(m + k)$ is possible [Epp98]. We aimed to write an implementation of Eppstein's algorithm in Python. Specifically, we wanted to write it in such a way that the algorithm could take as input a DAG from the `networkx` library.
Eppstein's algorithm for finding the k shortest paths between a source and target vertex makes use of a shortest path tree, this is a sub-graph of the input graph that consists of the shortest paths from each vertex in the graph to the target vertex. It is tree structured with the target vertex as

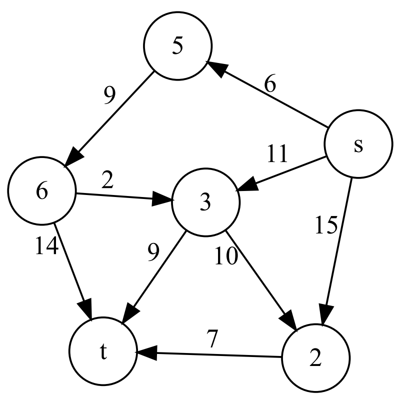
root. This algorithm does not take into account multiple paths that have the same weights but have different edges in the shortest path tree and assumes that the k shortest paths that use distinct non-shortest path tree edges *are* the k shortest paths (in practice this often corresponds to the k distinctly weighted shortest paths). As so, we provide a slight modification to the algorithm such that it will also consider the paths of the same weight with same non-shortest path edges. These paths will thus be longer than the shortest path but shorter than the next shortest path of additional weight.
Even though Eppstein's algorithm can achieve $O(m + k)$, this involves many additional complex steps. We here aim to implement and explain the base algorithm that runs in $O(m+n \log n+k \log k)$ as described in his paper [Epp98]. This is already better in terms of time complexity than the variant of Dijkstra's algorithm since instead of a multiplication of the factor $n \log n$ with k we only add a term of $k \log k$. Due to a shortage of time we were unable to implement the complete version but the base algorithm can serve as a basis for the complete version. As we only consider the layered graphs which are DAG's, we will not cover the treatment of cycles, the details on how cycles are handled can be found in [Epp98].
Eppstein's algorithm consists of six major steps in order to find the k shortest paths from source node s to target node t which involve so-called "sidetrack edges" with "sidetrack costs", these terms are further explained in 3.2.2. The six steps are listed below:

1. Finding the shortest path tree
2. Calculating sidetrack costs
3. Creating a heap of sidetrack edges on the shortest path from v to t for each vertex v
4. Forming a directed acyclic graph of the heaps created in step 3
5. Modification of the newly created graph
6. Heap creation from the modified graph

An explanation of the algorithm as well as notes on the implementation and additions to the original algorithm to account for multiple shortest paths of the same weight that use the same non-shortest path tree edges are described in the following six sections. The seventh section will describe how the heap that results from the sixth step can be used to obtain the paths of the graph. To aid the explanation we make use of an example graph shown in Figure 7, it is also a DAG.

3.2.1 Shortest path tree

The first step of the algorithm is to find the shortest path tree T from target node t, i.e. the shortest path from each vertex in the graph to t. This can be done by inverting the edges of the graph and performing Dijkstra's algorithm starting at t to find the minimal distance to each vertex in $O(m + n \log n)$. In DAG's this is reduced to $O(m + n)$ because when the vertices are traversed in topological order it is guaranteed that all incoming edges of a vertex have already been processed since there are no cycles. This means that each edge and each vertex are visited only once. Figure 8 shows the example graph from Figure 7 with the shortest path tree and the minimum distance from each vertex to t.



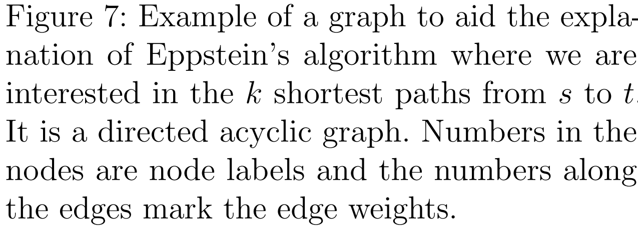


Figure 7: Example of a graph to aid the explanation of Eppstein's algorithm where we are interested in the k shortest paths from s to t. It is a directed acyclic graph. Numbers in the nodes are node labels and the numbers along the edges mark the edge weights.

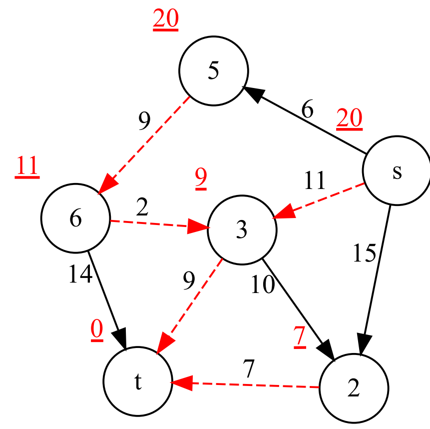


Figure 8: Example of shortest path tree with root t marked by dashed red edges. The shortest path distance from each node to t is underlined red.

3.2.2 Sidetrack costs

All edges that are not in the shortest path tree, i.e. all edges in $G - T$, are called "sidetrack edges". Given an edge e, the "sidetrack cost" $\delta(e)$ is the additional distance traveled when being "sidetracked" along e instead of taking a shortest path to t. It can be expressed as

$$\delta(e) = \ell(e) + d(\text{head}(e), t) - d(\text{tail}(e), t). \tag{19}$$

Figure 9 shows the previous example graph with sidetrack costs. Note that in a graph where all paths from s to t have unique weights, a sequence of sidetrack edges uniquely corresponds to a path from s to t, as the rest of the path is defined by the edges in T. Also note that a sequence of sidetrack edges may not correspond to any s-t path if a pair of sidetrack edges in the sequence cannot be connected through any path in T. The path weight that corresponds to the path in G described by a sequence of sidetrack edges can be obtained by adding the sum of all sidetrack costs of the edges in the sequence to the shortest path distance.
As an example, consider the sidetrack edge sequence $\{(s, 5), (3, 2)\}$. Starting from s we follow the sidetrack edge with additional cost 6 to vertex 5. From vertex 5 we follow the shortest path until the vertex that we visit corresponds to the tail of the next sidetrack edge in the sequence. We then follow the sidetrack edge that has additional cost 8. Finally, we again follow the shortest path until we arrive at t. The additional cost corresponding to this path is the sum of sidetrack costs of edges $(s, 5)$ and $(3, 2)$, which is 6+8=14. The total cost of the path in G that this sequence of sidetrack edges corresponds to is then the sum of the shortest path weight from s and the additional cost, which is 20+14 = 34.

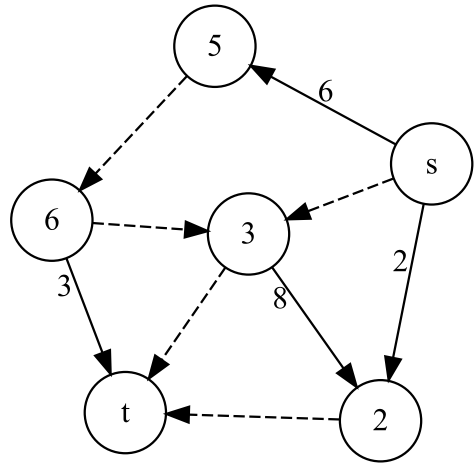


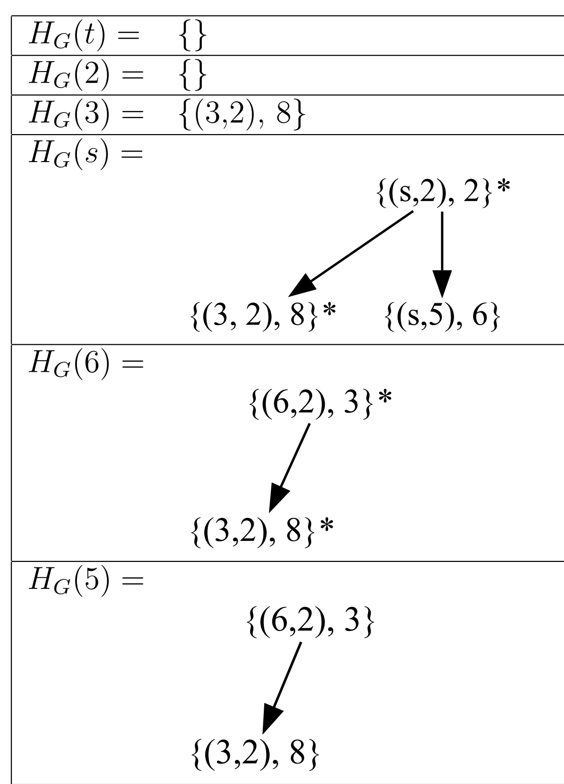
Figure 9: Example of graph with sidetrack cost on the sidetrack edges. Dashed edges mark the shortest path tree from t.

3.2.3 Sidetrack edge heaps for v

In the fourth step we create a heap $H_G(v)$ for each vertex v of all sidetrack edges with tails on the shortest path from v to t, ordered by sidetrack cost. These heaps will be used in the next step to create the directed acyclic graph. When finally traversing the graph they will represent possible options of sidetrack edges.
The first step in finding these heaps for each vertex v is creating a binary heap $H_{out}(v)$ of the sidetrack edges emanating from v, with the added restriction that the root only has one child, this is shown for the example graph in Table 2. Next, these heaps will be merged by applying a breadth first search (BFS) on the shortest path tree starting from t. If we look at H_G as a binary heap of H_{out} heaps maintained by the value of their root, then the merging is done by pushing the H_{out} heap of subsequent vertex onto the H_G heap of the current vertex to form the H_G heap of the subsequent vertex in the BFS. For the start node t we set H_G to be equal to H_{out}. For our example graph, the weight of the roots of the H_{out} heaps are 8, 2 and 3, so the H_G heaps will be maintained by these sidetrack edges. Additionally, each node on a path from the root of the H_G heap that is the root of an H_{out} heap that is updated by the insertion of a new H_{out} heap gets a unique identifier such that H_G is built in a persistent manner, this is important for the next step. The creation of the H_G heaps is shown in Table 3, the updating of nodes (and thus assignment of a unique identifier) is indicated with an asterisk.
An addition that was made to this procedure was the following: If there are multiple shortest paths leading to vertex v, say one coming from vertex u and one coming from w, then $H_G(v)$ will need to contain both $H_G(u)$ and $H_G(w)$. It can however not be obtained by applying the previous procedure once for u and once for w as $H_G(v)$ obtained from w would then simply overwrite $H_G(v)$ obtained from u which causes the nodes that are in $H_G(u)$ but not in $H_G(w)$ to be missing in $H_G(v)$. To make up for this problem, the heaps are merged by first applying the procedure as usual for vertex u and v and then popping each node from $H_G(w)$ and subsequently pushing it onto $H_G(v)$. If s_w and s_v denote the sizes of $H_G(w)$ and $H_G(v)$ (after merging with $H_G(u)$) then this procedure is done in $O(s_w \log s_w + s_w \log(s_v + s_w))$.

$H_{out}(t) =$	$\{\}$
$H_{out}(2) =$	$\{\}$
$H_{out}(3) =$	{(3,2), 8}
$H_{out}(s) =$	{(s,2), 2} {(s,5), 6}
$H_{out}(5) =$	$\{\}$
$H_{out}(6) =$	{(6,t), 3}

Table 2: H_{out} heaps of all the vertices in the example graph. (u, v) denotes the edge and is followed by its sidetrack cost. $\{\}$ marks an empty heap.



$H_G(t) =$	$\{\}$
$H_G(2) =$	$\{\}$
$H_G(3) =$	{(3,2), 8}
$H_G(s) =$	{(s,2), 2}* {(3, 2), 8}* {(s,5), 6}
$H_G(6) =$	{(6,2), 3}* {(3,2), 8}*
$H_G(5) =$	{(6,2), 3} {(3,2), 8}

Table 3: H_G heaps in visiting order of the BFS from top to bottom. An asterisk marks nodes on paths from the root that are updated by the last insertion of an H_{out} heap.

3.2.4 Graph creation

Now using the H_G heaps of the previous section and the path tree from t from section 3.2.1 a graph from which a bounded-degree heap can be found can be constructed. Note that H_G can be interpreted as heap with at most degree 3, where the root nodes of the H_{out} heaps have two pointers to other H_{out} heaps, and one pointer to its only child which connects it to the rest of the H_{out} heap. This is the reason that the H_{out} heaps had the added restriction that their root could only have one child. Let us now construct a directed acyclic graph $D(G)$ by finding all those heaps $H_G(v)$ that have v as head of the last sidetrack edge in a sequence of sidetrack edges. The vertices and edges in the graph are those of the found H_G heaps where the unique identifier of nodes is also taken into account such that unique nodes will be created when several H_G heaps contain nodes denoting the same sidetrack edge and it is necessary to distinguish them. See Figure 10, from here on we will denote the sidetrack edges by their sidetrack cost as all sidetrack edges in our example have unique sidetrack costs. This makes it easier to distinguish the different edges and visualise them.

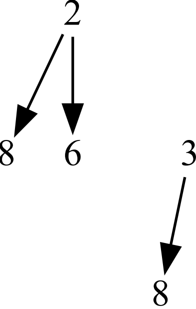


Figure 10: $D(G)$, a graph where each node has at most out degree 3. Essentially displaying $H_G(s)$ and $H_G(5)$. They are the corresponding heaps to the head of the last sidetrack edge of the empty sequence, which is just s and of sidetrack edge 6. The H_G heaps of the heads of sidetrack edges 8, 2 and 3 are empty.

The current implementation adds the H_G heaps of all the vertices in G to $D(G)$ by iterating over the vertices. This will not matter for the correctness as these will become unreachable or duplicate components of the graph later on but this implementation can be improved upon for maximum space efficiency. There was no opportunity to adjust this in due time.

3.2.5 Graph modification

From $D(G)$ we find the path graph $P(G)$, (Figure 11). $P(G)$ will contain the same vertices and edges as $D(G)$, its vertices are unweighted but edge weights are added. The weight of each edge e is $\delta(\text{head}(e)) - \delta(\text{tail}(e))$, these edges are called *heap edges*. Furthermore, edges are added from each node v in $P(G)$ to the node in $P(G)$ that corresponds to the root of $H_G(w)$ when w is the head of v in $G - T$, these edges are called *cross edges*. For our example graph, the edge with sidetrack cost 6 points to vertex 5, so in $P(G)$ we add a cross edge from node 6 to the root of $H_G(5)$ which is (6,2) with sidetrack cost 3. Sidetrack edges with sidetrack cost 2, 8 and 3 all point to vertices that

have empty H_G heaps, so for those no cross edges are added. The cross edges are given the weight equal to the sidetrack cost of the sidetrack edge that they point to. Lastly, a root r is added with edge pointing to the node that corresponds to the root of $H_G(s)$, this edge is weighted the same way as the cross edges.
$P(G)$ is now a DAG in which each vertex has at most out-degree four. Each path in $P(G)$ rooted at r corresponds to a sequence of sidetrack edges and as such to an s-t path in G. The weight of each such path in $P(G)$ rooted at r corresponds to the extra distance traveled on top of the shortest path.

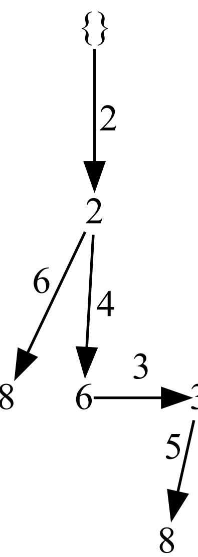


Figure 11: Path graph $P(G)$. The nodes represent sidetrack edges, the root represents no sidetrack edge. The edges have the difference of the sidetrack edge cost of their head and tail as weight. The edge from 6 to 3 is a *cross edge* and has weight equal to the head of the cross edge.

3.2.6 Heap creation

The next step is to create a heap from the paths in $P(G)$, each path in $P(G)$ will form a node in the path heap $H(G)$. To obtain a 4-heap $H(G)$ from $P(G)$ in which the nodes implicitly represent paths in G through their sidetrack edge sequence, we perform a BFS starting at root node r. Throughout the BFS we keep track of a sequence of sidetrack edge nodes and the total weight of the path over which we have traveled so far. r corresponds to the empty sequence and thus the shortest path. When we travel over a cross edge we add the tail of the cross edge to the sequence. When we visit a vertex v in $P(G)$, we add a node to $H(G)$ with v added to the sequence and the total distance traveled from r to v, though the sequence used in the BFS stays unchanged. When adding this node to $H(G)$, we can place it as child of its predecessor in the BFS, see Figure 12.
From $H(G)$ an ith shortest path that has unique sidetrack edges can be obtained in $O(\log i)$, k such shortest paths can be found in $O(k \log k)$.
As this does not account for the k shortest paths when some of the k shortest paths have the same sidetrack edges (thus also the same weight), another adjustments was made to the algorithm. After

having popped from $H(G)$, all paths of same weight for the just obtained sequence are found. This is done by iterating over the sequence. For each path in T found between a pair of sidetrack edges in the sequence we find all the paths in-between the subsequent pair of sidetrack edges. Note that this will take exponential time. When we take this into account, the original time complexity will be violated. The worst case for the layered graphs that we work with will arise when all edges have the same weight. In that case all paths will be found from s to t which will take $O(2^n)$ or more precisely $O(q^\ell)$ in terms of ℓ layers and p points per layer.

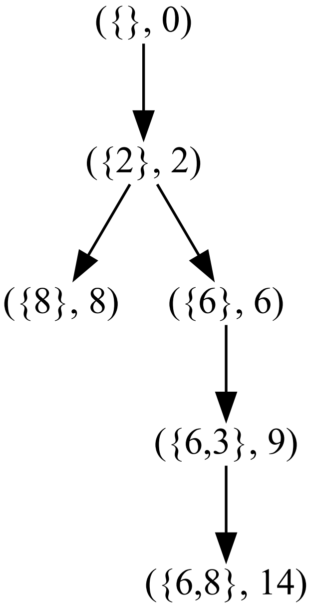


Figure 12: The final heap for our example graph with a bounded degree of at most 4. Each node is a pair of a sidetrack edge sequence and additional weight with respect to the shortest path.

3.2.7 Obtaining the paths from the heap

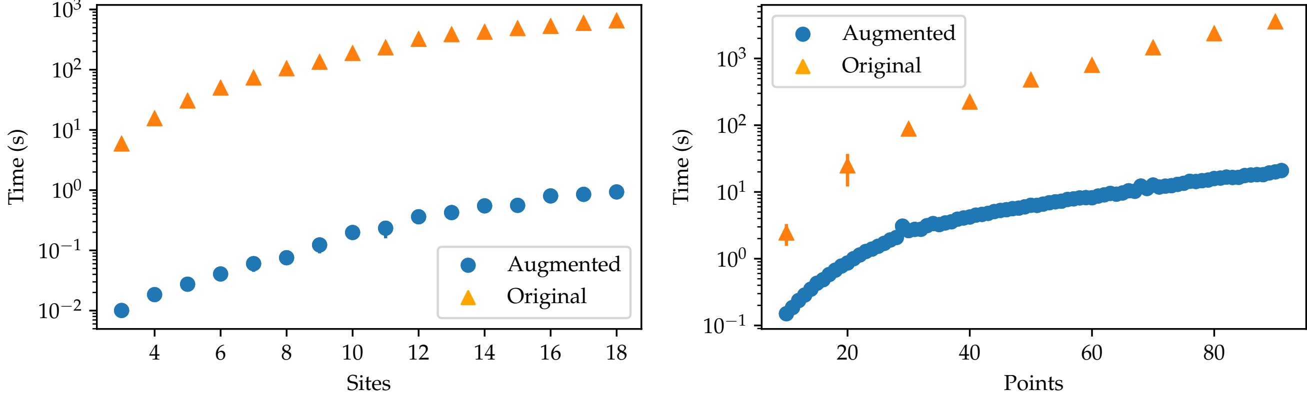
Finally, to obtain the k paths in G from the 4-heap $H(G)$, we pop k times from $H(G)$. Each node that we pop contains a sequence of sidetrack edges, which is an implicit representation of a path in G. Because of the heap structure of $H(G)$ the weight of the path represented by every node that is popped next will be larger or equal to the path weight of the nodes that have already been popped. Each node will also contain an additional sidetrack cost and the total weight of the path can be computed in constant time by adding the additional sidetrack cost to $d(s,t)$, (the shortest path distance). The edges that the path consists of explicitly can be found in time proportional to the number of edges by starting a traversal at the source vertex s and following the edges in the shortest path tree T and edges in the sidetrack edge sequence, as explained in section 3.2.2.

4 Results

To test these methods, six benchmarks were run that can be divided into two categories. The first category benchmarks the graph augmentation method. The second category benchmarks our implementation of Eppstein's algorithm. In all of the benchmarks we are interested in the run time of the different methods for different graph configurations. Because run times may vary depending on the computer hardware that is used, we list the hardware for our experiments in the appendix. Specifically, the processor that is used and the amount of available memory and its speed will influence the speed at which calculations can be performed. We will refer to the used hardware in each experiment, this way it is easier to make comparisons with experiments that are run on different hardware in the future.

4.1 Graph Augmentation

To test the graph augmentation method, two benchmarks were run. For both benchmarks we made use of the variant of Dijkstra's algorithm for the k shortest paths. Both benchmarks compare the original method of iterating over all start and end node pairs with the method of running the algorithm once on the augmented graph. For the first experiment we use a fixed number of points for each site and vary the number of sites, see Figure 13a. For the second experiment we choose a fixed number of sites and vary the number of points per site, see Figure 13b. The benchmarks time the run time of the algorithm for all of the different graph configurations.



(a) All graphs had 100 points per site.

(b) All graphs had 100 sites for each number of points.

Figure 13: Comparison of original and augmented graph method showing the run time of the variant of Dijkstra's algorithm for the k shortest paths. For each data point, 5 randomly weighted graphs were timed to get an average run time. For most data points, the error bars are smaller than the symbol size.

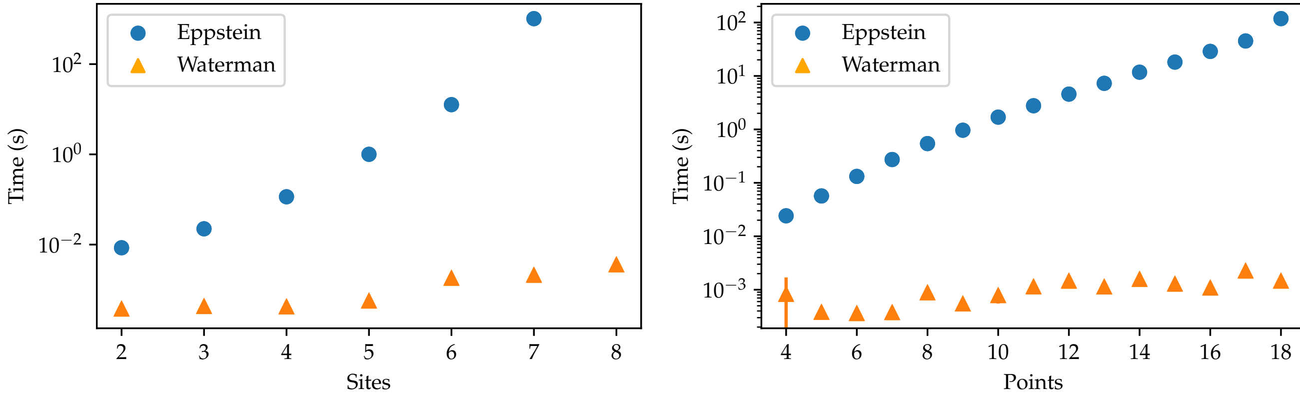
In Figure 13a we see two trends that curve in roughly the same way. We see that the difference in runtime grows when we increase the number of sites in the graph and keep the number of points the same. Figure 13b shows two trends on log scale and we can see how the trends for the original and augmented graph slowly diverge, this represents the quadratic speedup of q^2 as we now increase the number of points q and keep the number of sites the same.
All benchmarks for Graph Augmentation were run on the same machine A.1.

4.2 Eppstein's algorithm

To test the performance of our implementation of Eppstein's algorithm we ran four more benchmarks, these can also be divided into two categories. Two of the benchmarks were run on graphs with random weights, and the two others were run on graphs with weights defined by the Ising Hamiltonian. In the last category, small random perturbations were added to the weights such that the k shortest paths would be distinctly weighted and thus uniquely defined by sidetrack edge sequences. This was done so that the algorithm would still run in the time complexity of Eppstein's algorithm and we would not see an effect of the exponential time needed to calculate paths with same sidetrack edge sequence. By doing this we could in the future also test the algorithm on the graphs without perturbation and see how much time is spent on calculating paths with same sidetrack edge sequences. All of the four tests compare Eppstein's algorithm to Waterman's algorithm in terms of run time. The random weights and perturbations were seeded such that both algorithms ran on the same random graphs.

4.2.1 Randomly weighted graphs

For the randomly weighted graphs two tests were run. One uses a fixed number of 9 points for each site and varies the number of sites, see Figure 14a. The other uses a fixed number of sites and varies the number of points per site, see Figure 14b. We used a number of 9 points per site in 14a because during the testing we found out that the implementation of Waterman's algorithm can not handle larger graphs as it reaches the maximum recursion depth in Python. Waterman's algorithm could be rewritten in a non-recursive fashion to run tests for a higher number of points, there was no time for this however. In 14a we also see that the data point for Eppstein at 8 sites is missing, this is because the algorithm was running for over 12 hours for the 5 repetitions it performs, meaning that each run was roughly taking over 2.4 hours or over 8500 seconds.



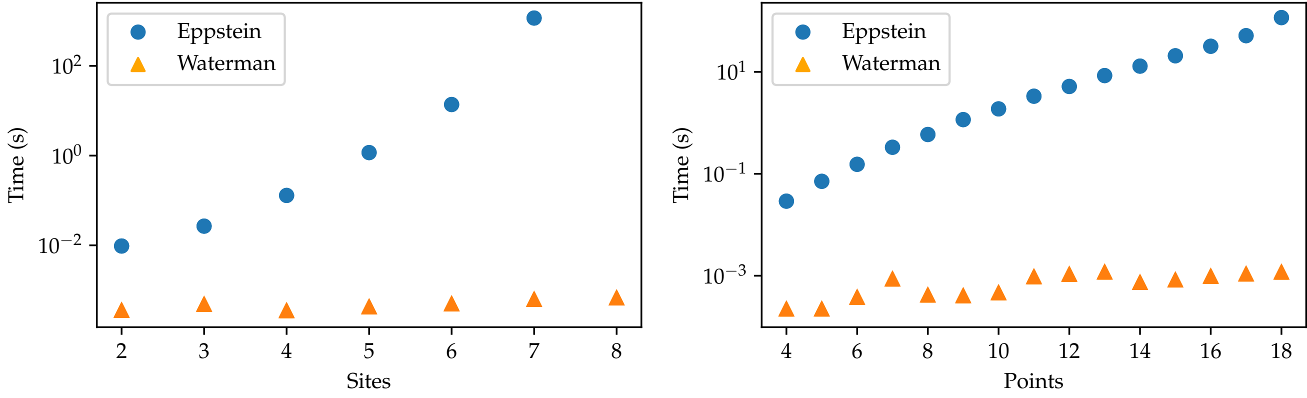
(a) All graphs had 9 points per site.

(b) All graphs had 5 sites for each number of points.

Figure 14: Comparison of Eppstein's and waterman's algorithm on graphs with randomly initialised edge weights. 5 repetitions were done for each execution and the average and standard deviation are plotted. For most data points the error bars are smaller than the symbol size.

4.2.2 Slightly perturbed graphs

For the graphs in which the weights are defined by the Hamiltonian the same set of tests was performed as for the randomly weighted graphs. One uses a fixed number of points for each site and varies the number of sites, see Figure 15a. The other uses a fixed number of sites and varies the number of points per site, see Figure 15b. In Figure 15a we use a fixed number of 9 points and have 1 missing point for Eppstein for the same reason as in the previous section.



(a) All graphs had 9 points per site.

(b) All graphs had 5 sites for each number of points.

Figure 15: Comparison of Eppstein's and Waterman's algorithm on graphs with randomly perturbed edge weights. 5 repetitions were done for each execution and the average and standard deviation are plotted. For most data points the error bars are smaller than the symbol size.

All benchmarks for Eppstein's algorithm were run on the same machine A.2.

4.2.3 Eppstein analysis

The trends for the similar experiments of section 4.2.1 and 4.2.2 look the same, so we will cover their analysis in the same section. We can see that in the experiments that vary the number of sites, figures 14a and 15a, Eppstein's algorithm seems to scale super-linearly on logarithmic scale. In the figures 14b and 15b we also see how our implementation of Eppstein seems to scale linearly on logarithmic scale which means it scales as a constant to the power of the number of points. This is not what we would expect given the time complexity of Eppstein's algorithm. After running the experiments, we timed all major steps of our implementation of Eppsteins algorithm and found out that there was still a mistake in the implementation. The function that was responsible for translating the path graph to a 4-heap as described in section 3.2.6 was implemented with a BFS that does not mark the nodes. This way the function explores all possible paths in the path graph which scales exponentially in the number of nodes of the path graph ($O(2^n)$). After addressing this mistake, we tested the code again but found that the implementation did not always provide us with the correct result anymore, which leads us to believe there must be a mistake earlier in the implementation. We have not been able to investigate this issue further.
What we would expect from a correct implementation of Eppstein's algorithm is that it scales better than Waterman's or Dijkstra's algorithm but that it might start off higher due to a large

prefactor. Eppstein's algorithm involves the manipulation of complex data structures and due to this, as noted by [AHN+15] as well, there is a large constant factor hidden in its O-notation.

5 Conclusion

We explored two possibilities of improving the run time of the new iterative method by J. Tura and P. Emonts for finding the product state approximations of ground states and their corresponding energy. The possible improvements were tested on randomly weighted graphs and graphs defined by the Ising Hamiltonian with small perturbations and compared to previous methods. The method that makes use of augmented graphs performs better than the method that iterates over all start and end node pairs. We were unable to find conclusive evidence on whether Eppstein's algorithm will provide a better run time on the graphs of interest due to the mistakes that remained in our implementation of his algorithm. In the future, it would be interesting to benchmark a correct implementation of the algorithm again against other existing algorithms to find where the crossover point lies and to see at what graph sizes the complexity of Eppstein's algorithm takes over and performs better than the other algorithms.
Further research may be done in improving the additions that were made to the algorithm to account for multiple paths of the same weight. The first addition was the merging of H_G heaps when multiple shortest paths lead to the same vertex from section 3.2.3. It might be possible to improve the complexity of this to $O(s_v + s_w)$ by implementing a heapify operation. The second addition was finding all paths for a given sequence of sidetrack edges whilst popping from the final heap of paths from section 3.2.6. We might improve this method by implementing a DFS that returns at most k paths for a sequence rather than all of them.

References

[09] Smite-Meister . Bloch sphere, January 2009.

[AAI10] Dorit Aharonov, Itai Arad, and Sandy Irani. An Efficient Algorithm for approximating 1D Ground States. *Physical Review A*, 82(1):012315, July 2010. arXiv:0910.5055 [quant-ph].

[AGIK09] Dorit Aharonov, Daniel Gottesman, Sandy Irani, and Julia Kempe. The Power of Quantum Systems on a Line. *Communications in Mathematical Physics*, 287(1):41–65, April 2009.

[AHN+15] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, February 2015. ISSN: 2374-3468, 2159-5399 Issue: 1 Journal Abbreviation: AAAI.

[Bel12] Ori Belkind. *Physical Systems: Conceptual Pathways between Flat Space-time and Matter*. Springer Science & Business Media, February 2012. Google-Books-ID: Wo1kuGWhcEcC.

[Blo46] F. Bloch. Nuclear Induction. *Physical Review*, 70(7-8):460–474, October 1946. Publisher: American Physical Society.

[Epp98] David Eppstein. Finding the k Shortest Paths. *SIAM Journal on Computing*, 28(2):652–673, January 1998. Publisher: Society for Industrial and Applied Mathematics.

[Gal99] Giovanni Gallavotti. *Statistical Mechanics*. Springer, Berlin, Heidelberg, 1999.

[GKSV03] Daniel Gottesman, A. Yu. Kitaev, A. H. Shen, and M. N. Vyalyi. Classical and Quantum Computation. In *The American Mathematical Monthly*, volume 110, page 969, December 2003. ISSN: 00029890 Issue: 10 Journal Abbreviation: The American Mathematical Monthly.

[Gri05] David J. (David Jeffery) Griffiths. *Introduction to quantum mechanics*. Upper Saddle River, NJ : Pearson Prentice Hall, 2005.

[Mer98] Eugen Merzbacher. *Quantum Mechanics*. John Wiley & Sons, January 1998. Google-Books-ID: 6Ja_QgAACAAJ.

[Mes66] Albert Messiah. *Quantum Mechanics*. North-Holland Publishing Compan, 1966. Google-Books-ID: n0xGzQEACAAJ.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010.

[SC10] Norbert Schuch and J. Ignacio Cirac. Matrix Product State and mean field solutions for one-dimensional systems can be found efficiently. *Physical Review A*, 82(1):012314, July 2010. arXiv:0910.4264 [quant-ph].

[WB85] Michael S. Waterman and Thomas H. Byers. A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Mathematical Biosciences*, 77(1):179–188, December 1985.

[Wil10] Colin P. Williams. *Explorations in Quantum Computing*. Springer Science & Business Media, December 2010.

A Computer Hardware

Hardware used in the experiments for reference.

A.1 Computer A

- `CPU: i7-8750H @3.9 GHz`
- `RAM: 16GB @2666 MHz`

A.2 Computer B

- `CPU: Ryzen 7 5800H @4.2 GHz`
- `RAM: 24GB @3200 MHz`