Quantum Information Processing (2023) 22:357
https://doi.org/10.1007/s11128-023-04115-w

Using an A^*-based framework for decomposing combinatorial optimization problems to employ NISQ computers

Simon Garhofer[1] · Oliver Bringmann[1]

Received: 12 December 2022 / Accepted: 10 September 2023
© The Author(s) 2023

Abstract
Combinatorial optimization problems such as the traveling salesperson problem are ubiquitous in practical applications and notoriously difficult to solve optimally. Hence, many current endeavors focus on producing approximate solutions. The use of quantum computers could accelerate the generation of those approximate solutions or yield more exact approximations in comparable time. However, quantum computers are presently very limited in size and fidelity. In this work, we aim to address the issue of limited problem size by developing a scheme that decomposes a combinatorial optimization problem instance into arbitrarily small subinstances that can be solved on a quantum machine. This process utilizes A^* as a foundation. Additionally, we present heuristics that reduce the runtime of the algorithm effectively, albeit at the cost of optimality. In experiments, we find that the heavy dependence of our approach on the choice of the heuristics used allows for a modifiable framework that can be adapted case by case instead of a concrete procedure.

Keywords Approximation · Combinatorial optimization · Problem decomposition · Traveling salesperson problem

1 Introduction

Quantum computers (QCs) in the noisy intermediate scale quantum (NISQ) era are predominantly restricted by their size, low coherence times and readout error. Therefore, the number of feasible algorithms and the problem sizes they can process are small

✉ Simon Garhofer
simon.garhofer@uni-tuebingen.de

Oliver Bringmann
oliver.bringmann@uni-tuebingen.de

[1] Embedded Systems, University of Tübingen, Sand 13, 72076 Tübingen, Baden-Württemberg, Germany

Published online: 27 September 2023

for NISQ machines [1]. However, there are approaches that promise early quantum advantage for combinatorial optimization problems such as quantum annealing and a gate-based equivalent called quantum approximate optimization algorithm (QAOA) [2, 3]. While there is not necessarily a runtime advantage, the approximation quality could potentially surpass classical algorithms in certain use cases.

1.1 Contribution

With the goal of harnessing those advantages more effectively, we propose a hybrid approach that is capable of decomposing a given large combinatorial optimization problem into smaller subproblems that can be solved on a quantum machine. For this paper, we specifically consider the traveling salesperson problem (TSP), as it is a widely known and well-researched problem with a multitude of practical applications [4]. However, this approach can be applied to other problems of combinatorial nature as well. Core of our approach is an algorithm based on the A^*-algorithm that is classically used for the search of shortest paths on a graph [5]. In our case, the nodes on the graph depict subproblems and their respective solutions. The optimum solution for the given large problem is thereby constructed by finding the best path through those subproblems.

The main advantages of this approach are the interchangeability of the methods that solve the subproblems and the theoretical optimality of the scheme as long as the subproblems are solved optimally and no further alterations with runtime optimization in mind are performed. Moreover, all generated subproblems are of the same size, suggesting the use of custom optimized circuits or dedicated hardware for problems of specific size. If the requirements for optimality are not strict, optimizations can be made that speed up the runtime effectively, albeit degrading the solution to an approximation.

We demonstrate our approach by presenting a thorough implementation for the TSP. We give an elaborate description of the TSP decomposition using our framework and introduce optimizations to acquire approximations in feasible time. Such optimizations include heuristics that guide the algorithm toward faster convergence, of which we propose several choices for our TSP example. Additionally, we adapt existing TSP tour length estimation techniques to this use case.

1.2 Related work

Earlier approaches with a similar goal include [6], where the problem is coarsely grouped and solved on those groups. Depending on whether a specific group is part of the solution, that group is slowly dissolved until a concrete optimal solution is found. By stowing away parts of the problem that are unlikely to be part of the optimal solution into higher level groups, the effective problem size is reduced considerably. However, the generated subproblems are always of different size, so the feasibility for a given quantum machine can't be guaranteed.

The approach in [7] heuristically reduces TSP instances, solves the reduced problem and reintroduces removed points and edges in order to retrieve the original instance.

Even though the problem sizes can be reduced effectively, it faces similar drawbacks for our use case as [6] in that it can't guarantee a certain problem size and it does not merge different solutions into a combined one.

In [8], the goal is very similar to ours in that large combinatorial problems are recursively decomposed into smaller problems in order for them to be solved on a quantum annealer. This decomposition is accompanied by heuristics that choose the graph nodes at which a "split" is performed and an objective function maximum/minimum value update which is used for pruning redundant subgraphs. While feasible for certain problems, the expected runtime for a TSP using this method is substantially worse than our approach.

1.3 Preliminaries

Combinatorial optimization problems in general and the TSP in particular are *NP-hard* problems; hence, there are no algorithms known that return optimal solutions for this class of problems efficiently, i.e., in a feasible amount of time for growing problem instance sizes. In the worst case, algorithms that compute optimal solutions have to check every possible solution given in the problem statement. In most of the practically relevant instances, approximation algorithms are used which compute solutions in considerably more favorable runtime at the cost of optimality [4].

Let $G = (V, E)$ be a complete undirected graph with vertices $V, |V| = n$ and edges $E, |E| = \binom{n}{2}$ and let $c : E \rightarrow \mathbb{R}$ be a function that maps each edge to some cost. The TSP asks to find a tour that visits all vertices exactly once and returns to the origin vertex while the cost of the tour, determined by c, is minimal [4].

A minimum Hamiltonian path (MHP) poses a similar problem without the postulation to return to the start of the tour. From now on, we will refer to the MHP as a variant of that problem, where the start and end point of the tour are fixed.

A^* is a greedy algorithm that finds a minimum path through a weighted graph using a priority queue and a heuristic function. The priority of each entry in the queue is described by

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the exact path length from the start to node n and $h(n)$ is the estimated path length from node n to the goal. The combined length $f(n)$ thus encodes an estimated path length incorporating information about the graph that has already been gathered. Consequently, $f(n)$ becomes more precise the closer n is to the goal. Using that value in a priority queue implements an algorithm that always takes the most promising node first and corrects itself along the way [5].

QAOA is a hybrid algorithm for optimizing combinatorial optimization problems. The basic QAOA setup as described in [3] consists of a parametrized quantum circuit that is iteratively optimized using classical algorithms. In the quantum circuit, an operator C that implements the cost function is, alternately with a mixer operator B, applied to a uniform superposition of all basis states, dependent on angles $\gamma_1, ..., \gamma_p, \beta_1, ..., \beta_p$:

$$|\gamma, \beta\rangle = e^{-i\beta_p B} e^{-i\gamma_p C} \ldots e^{-i\beta_1 B} e^{-i\gamma_1 C} |+\rangle$$

The expected value $\langle\gamma, \beta|C|\gamma, \beta\rangle$ is then minimized/maximized by some classical optimization algorithm.

The procedure described in [9] extends QAOA by constructing the mixer in such a way that given problem boundary conditions are adhered to. Constraints, such as "every city has to be visited exactly once," are therefore guaranteed to be met whereas in standard QAOA such constraints would have to be implemented via penalty terms in C. We propose an efficient implementation of such a mixer further below.

2 Decomposition algorithm

We will now describe the algorithm first on a high level, applicable to any combinatorial problem, and then a specific implementation for the TSP that we will further investigate in subsequent sections.

For an overview of the basic idea, see Fig. 1. The algorithm is based on A^*, where the individual nodes that A^* traverses are composed of a set of points given in the problem, for example cities in the TSP or items of different weight and value in the Knapsack problem. A^* is run on a classical computer and chooses what subproblems are to be solved on a quantum computer. The returned solution from the quantum machine in turn guides A^* toward the next subproblem.

In Step (1), *initial nodes* for the search graph are being generated. Any combination of points (e.g., cities in the TSP) is a valid initial node and represents a subproblem of manageable size for the given quantum machine. However, generating all existing subproblems is usually neither feasible nor required, since many of the theoretically generatable subproblems are not expedient for computing the final solution. A good heuristic for generating initial nodes is therefore required. Some options for initial node generation are discussed in the TSP case in Sect. 2.1. A different combinatorial problem would require a different opening heuristic.

Afterward, all initial nodes are rated concerning their probable value for a final solution in Step (2). This estimation is based on the subproblem itself as well as the set of remaining points not present in the node. The combination of those estimations serve as the heuristic function $h(n)$ in A^*. For the TSP implementation, we employed a fully connected neural network to that task. It is more closely described in Sect. 2.2.

Step (3) depicts the solution of a chosen subproblem that most likely leads to a good overall result. This solution is computed on a quantum machine, since that is the only remaining part of the problem with combinatorial complexity. The specific quantum algorithm used can be chosen freely by the user. Thereafter, subsequent nodes are generated that incorporate that already computed subsolution, which is indicated by the rectangle holding previous subproblem elements in a certain order. Until a final solution is found, Step (2) and (3) loop. Hence, an estimation of all newly generated subproblems would follow. Analogously to Step (1), there are exponentially many possibilities for followup nodes. A heuristic that is not necessarily equivalent to the one used in Step (1) is consequently required. For the TSP, we used a heuristic incorporating the nearest neighbors of a point, see details in Sect. 2.1.

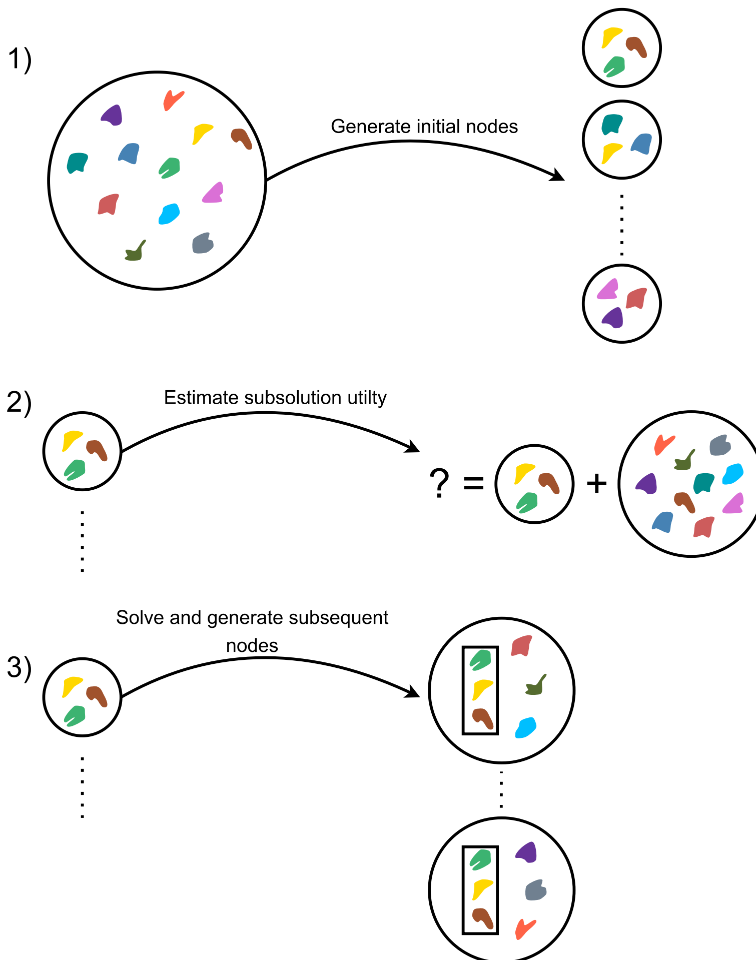

Fig. 1 High level depiction of our proposed algorithm

Note that in this setup any black box capable of solving a given subproblem is acceptable and is thus not strictly required to be a QC. However, only QCs are capable of profiting from the approach we propose, since classical computers don't suffer from such rigorous resource limitations that they would require problem instances to be partitioned in this way.

The main advantage of having a fixed subinstance size k is that the solution process on a given quantum machine can be optimized without concern for generality. In other words, the maximum capabilities of a quantum machine can be utilized for solving the combinatorial problem while the classical container guides every step of the way. As long as varying noise, connectivity, qubit counts, etc. have to be considered when building a quantum circuit, the ability to customize such a circuit to those conditions is essential.

We will now describe an implementation for the TSP. Here, the individual nodes that are traversed by A^* are composed of a set of points given in the TSP that have already been visited and a set of points, which we call *support points*, that act as equidistant fixed points on the path. The subproblems that are generated always connect two of those support points.

Let n be the number of points for the given TSP instance and k be the subinstance size that the QC is capable of solving. Initially, the procedure starts with a collection of nodes that contain an empty set of visited points and a $\frac{n}{k}$ sized set of support points that still need to be visited. Points in the latter set will be placed on the explored path at time steps $k, 2k, \ldots, \frac{n}{k} \cdot k$. The easiest way to generate those initial nodes is therefore to consider all $\binom{n}{\frac{n}{k}}$ sets of support points. This however quickly becomes infeasible, especially for problems where many intermediate steps should be taken. We discuss some options for generating initial nodes in Sect. 2.1.

Likewise, in the *goal nodes* we are trying to reach the set of visited nodes corresponds to the full set of TSP points, whereas the set of support points is empty.

For A^* to work, there needs to be some method in place that estimates the distance from the current node to the goal node (i.e., the function h). In our case, this means estimating the optimum tour length of a TSP for all initial nodes and the optimum tour length for an MHP for all subsequent nodes. The distinction between initial nodes and the rest is made due to the tour being estimated in all intermediate steps is incomplete. Hence, those estimations need to be made starting from the last point of the previously computed path and ending in their first. Moreover, all estimations need to consider the positions of the given support points, since their choice has significant impact on tour length. Approaches for generating those estimations are discussed in Sect. 2.2. For now, we assume estimations as given.

After estimations for all initial nodes are computed, the most promising one is picked first. This node is then “expanded”: New nodes are generated from the set of support points in the chosen initial node. Each expansion node fixes a single support point as the first one to be visited and leaves all others in a non-determined order. Additionally, in order for some solver to be able to create a solution, the sets of points that are visited between the start/end point and the first support point are defined at that time as well. The tour length estimation for the expansion nodes is the sum of the MHP estimation from the start point to the fixed first support point and the MHP estimation from the first support point to the start point, the latter considering all remaining support points. Analogously to the generation of initial nodes, simply generating nodes for all $\binom{n-\frac{n}{k}}{k}$ combinations of points is infeasible for practical use. An intuitive method for choosing intermediate points is the usage of a points nearest neighbors, which is more closely described in Sect. 2.1.

In the iteration thereafter, if another initial node is chosen, the procedure is the same as described in the paragraph above. If however, an “expanded” node is chosen, the subproblem given by the start point, the first support point and the k intermediate points is solved by the subroutine on the QC. Afterward, several options arise for the generation of more subsequent nodes.

If there are at least k points that have not been visited yet, the node is expanded again; hence, all remaining support point and intermediate point combinations become

subsequent nodes. Analogously to the case for initial nodes, the optimum tour length estimation is a combination of estimating an optimum MHP tour length for the intermediate points and estimating such a tour length for an MHP with the remaining support points at equidistant time steps.

If all TSP points were visited, a single new node is inserted where the estimated remaining tour length is simply the cost of the edge between the final point in the current path and the start point.

An example execution of the algorithm is shown in Fig. 2. In that example, an instance with $n = 13$ and $k = 4$ is shown; hence, there have to be 3 support points on each complete path. We fix one point as the starting point, such that the remaining 12 points are divisible by the step size $k = 4$.

(a) is the initial state where all initial nodes are represented by a curved line from the start to the goal point and the support points on top.

In (b), one of these nodes is chosen through the estimated length of the optimum tour through its listed support points.

(c) depicts the expansion of the chosen node, where each support point can come first in the tour and each candidate from CANDIDATES_STEP can be between the start node and the first support node. The straight, thicker line represents a tangible MHP instance.

In (d), one of the expanded nodes is chosen and the problem determined by the start node, $M_{Problem}$ and the first support node is solved.

(e) shows the solved subproblem as purple line and the subsequent expanded node, where each of the remaining support points and candidates form a new path to the goal while the computed solution from before is fixed. The set of visited points is updated on that path.

In (f), an expanded node on the path is chosen, resulting again in a solved subproblem and a node expansion in (g).

(h) illustrates that the path that was previously favored is at that point estimated worse than a different path. The subinstance on that path is solved and the subsequent node expanded in i). The visited points and support points from before remain while the subproblem points are different.

(j) shows a state where the goal node can be reached after the chosen subproblem is solved. The estimation in k) is hence simply the edge cost back to the start.

In (l), the solution to the problem is completed.

The exact procedure is depicted in Algorithm 1 as pseudocode. Therein, parameter e is a constant factor that is multiplied with all estimated tour lengths in order to keep the estimations admissible. We will call this factor *error margin factor* in the following sections. A python implementation of the algorithm is available at.[1] In practice, some branch operation needs to be in place for problems where $n - 1$ is not divisible by k. In such cases the last step will consist of subproblems of size $k' < k$, which is still feasibly solvable on a QC.

[1] https://github.com/ekut-es/decomposition-for-quantum.

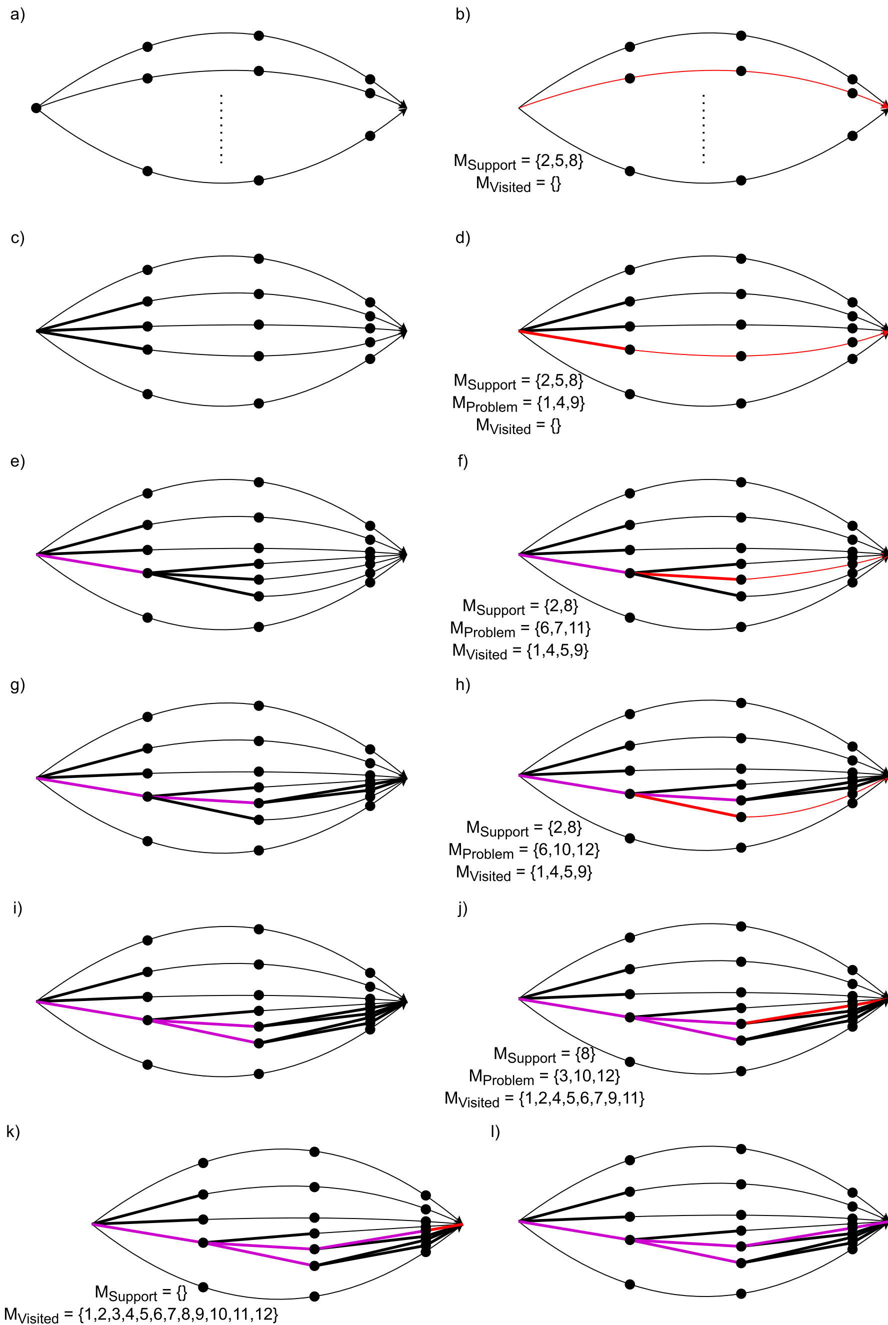

Fig. 2 Example execution of the proposed procedure with 13 TSP points

Algorithm 1 TSP_Astar_partition(P, A, k, e)

```
1: n ← |P|
2: M ← {1, ..., n}
3: queue ← PRIORITY_QUEUE(∅)
4:
5: // Obtain an initial set of queue nodes
6: for all M_s ∈ CANDIDATES_INIT(M, k) do
7:     d_e ← ESTIMATE_TSP_SUPPORT(M_s)
8:     d_e ← d_e · e
9:     queue.put(d_e, (0, 0, {}, (), 0, M_s, {}))
10: end for
11:
12: while not queue.empty() do
13:     // Values in the node queue are a tuple with the following components:
14:     // 1. Start node of the next subproblem
15:     // 2. End node of the next subproblem
16:     // 3. All nodes that are already visited
17:     // 4. The currently existing path
18:     // 5. The currently already traveled distance
19:     // 6. All remaining support nodes
20:     // 7. The set of nodes to be used in the next subproblem
21:     expansion ← queue.get()
22:     m_s^0 ← expansion[0]
23:     m_n^0 ← expansion[1]
24:     M_v^0 ← expansion[2]
25:     p^0 ← expansion[3]
26:     d^0 ← expansion[4]
27:     M_s^0 ← expansion[5]
28:     M_t^0 ← expansion[6]
29:
30:     // Initially, there is no target set; hence, no subproblem is solved
31:     if m_s = 0 and m_n = 0 then
32:         p^1 ← ()
33:         d^1 ← 0
34:         M_v^1 ← {}
35:     else
36:         // If final node is reached, terminate
37:         if m_n = 0 then
38:             return p^0 + (0), d^0 + A[m_s, 0]
39:         end if
40:         // Else, take a step of size k toward the goal
41:         // Insert some MHP solver here
42:         p_t, d_t ← SOLVE_SUBPROBLEM(m_s^0, M_t^0, m_n^0)
```

2.1 Node generation heuristics

The number of priority queue nodes that are generated in each step can be reduced considerably by using a nearest neighbor heuristic. Instead of having CANDIDATES_STEP be an iterator for all combinations of remaining points replace it with an iterator that returns each remaining point and its k closest neighbors. That way the iterator returns only $\mathcal{O}(n)$ candidates per step.

The intuition here is that good tours don't pass through the given area multiple times. Candidates where points are on opposite sides of the graph are therefore not

```
43:         p^1 ← p^0 + p_t
44:         d^1 ← d^0 + d_t
45:         M_v^1 ← M_v^0 ∪ {m : m ∈ p_t}
46:     end if
47:
48:     if M_v^1 = M then
49:         // Special case for when only the return to the start node remains
50:         queue.put(d^1 + A[m_n, 0], (m_n, 0, M_v^1, p^1, d^1, {} {}))
51:     else
52:         for all m_n^1 ∈ M_s^0 do
53:             for all M_t^1 ∈ CANDIDATES_STEP(M \ M_v^1 \ (M_s^0 \ {m_n^1}), k) do
54:                 if m_n^1 ∈ M_t^1 then
55:                     M_r ← M \ M_t^1 \ M_v^1
56:                     d_{e,t} ← ESTIMATE_MHP(m_n^0, M_t^1, m_n^1)
57:                     // Estimate the remaining tour length dependent on what
58:                     // problem size is ahead
59:                     if |M_r| < 1 then
60:                         d_{e,r} ← A[m_n^1, 0]
61:                     else if |M_r| = k then
62:                         d_{e,r} ← ESTIMATE_MHP(m_n^1, M_r, 0)
63:                     else
64:                         d_{e,r} ← ESTIMATE_MHP_SUPPORT(m_n^1, M_r, M_s^0 \ {m_n^1})
65:                     end if
66:                     d_e ← d_{e,t} + d_{e,r}
67:                     queue.put(d^1 + d_e, (m_n^0, m_n^1, M_v^1, p^1, d^1, M_s^0 \ {m_n^1}, M_t^1))
68:                 end if
69:             end for
70:         end for
71:     end if
72: end while
```

likely to be useful. Using k nearest neighbors emphasizes that the (sub-) tour stays compact and creating a candidate for each point ensures that every point is considered in that respective step. The quality of this heuristic increases with $\frac{k}{n}$ - a greater fraction of all points in a candidate should result in better tours simply because more points are considered at once.

The number of priority queue nodes that are generated upon initialization with CANDIDATES_SUPPORT could be reduced by a heuristic as well. While the nearest neighbor heuristic encodes closeness, the support nodes should be evenly spread to ensure that a solid fraction of the given area is covered. No heuristic of this kind has been implemented for this work, however. A possible heuristic could be to find for each pair of cities a set of cities where for each added city the average distance from the existing set is closest to the average pairwise distance in the set, reducing the initial number of nodes from $\binom{n}{\frac{n}{k}}$ to n^2.

A more complex approach is illustrated in Fig. 3. A circle with $\frac{n}{k}$ segments and some given width can be used to generate initial candidates by taking combinations of points, exactly one from each segment. Intuitively, this method should result in support point sets where the individual surrounding point sets don't overlap much while still retaining enough combinatorial leeway for optimization. The impact of different opening heuristics should be investigated in future work.

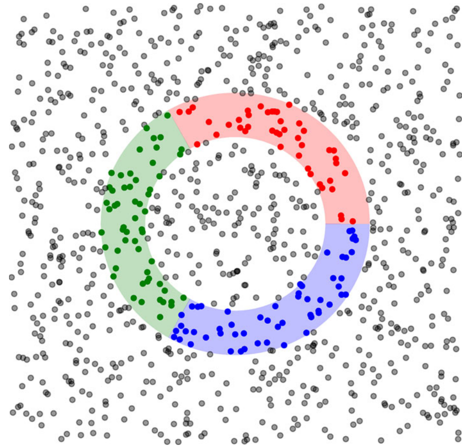

Fig. 3 Illustration of a suggested procedure to initially populate the priority queue of our A^* inspired process. The support point sets that are required could be constructed from combinations of points that lie on circle segments of the same size

2.1.1 Correctness

The algorithm finds an optimum solution if the subroutine that solves the small TSP instances returns optimum solutions, if no priority queue candidates have been filtered by heuristics and if the TSP tour length estimations never overshoots or is reliably rectified by an error margin factor.

Proof *(Sketch)* If no priority queue nodes have been filtered, all support point combinations and problem point combinations in between are considered; hence, the search graph is complete in the sense that the optimal solution is guaranteed to be contained.

If all combinations of support points are considered, all sequences of support points are considered through node expansion as well.

If all subproblems are solved optimally, a path through a given sequence of support points must be of optimal length.

Through the combination of these statements we can conclude that the optimum path can always be found through exhaustive search. If we accelerate this search with A^*, the optimality follows from the optimality of A^* in its tree search version because the tour length estimation is presumed to be admissible [5]. The tree property of the search graph follows from the lack of backward edges from bigger subsolutions to smaller ones and from the fact that all termination nodes are different from each other.

Conversely, if entries from the priority queue are removed through the use of some heuristic, the optimum solution is not guaranteed to remain in the graph. If the tour length estimation is not guaranteed to be admissible, the optimality of A^* no longer holds. □

2.1.2 Runtime

First consider the number of priority queue nodes that are generated. There are $\frac{n}{k}$ support points on any tour. Thus the number of initial nodes on the A^* priority queue

is $\binom{n}{\frac{n}{k}}$ in the combinatorial case. This number grows exponentially in $\frac{n}{k}$; hence, exponentially many TSP tour length estimations would need to be run. A simple heuristic that filters out many ill-fated candidates is therefore required.

For each visited node $\mathcal{O}(\frac{n}{k} \cdot \binom{n-\frac{n}{k}}{k})$ nodes are generated in the combinatorial case, which are exponentially many in k. Using the heuristic to only generate nodes for each city and their k closest neighbors brings this down to $\mathcal{O}(\frac{n}{k} \cdot n)$.

For this work, the estimation was done using small neural networks, see Sect. 2.2 for details. The estimation by the network can be done in constant time while the feature extraction takes linear time in our naive implementation. All initial nodes need to run 1, all subsequent nodes 2 estimations.

Now consider the number of priority queue nodes that are visited. In the best case, the number of visited nodes is $\frac{n}{k}$, in the worst case all of them, which could be, depending on whether or not heuristics were used, exponentially many. For each visited node, a small TSP instance is solved (approximately) on the quantum computer. If QAOA is used, each instance requires time $\mathcal{O}(k^2)$ [9].

The number of visited nodes depends heavily on the quality of the TSP tour length estimation. Especially if it overshoots regularly, a high error margin factor needs to be chosen, which inflates the number of visited nodes considerably. See Sect. 3 for experimental details.

2.2 Tour length estimation

Previous work on estimating optimum TSP tour lengths includes [10] where an arrangement of features of a TSP instance were used to find linear coefficients for different estimation models. Additionally, those features were used to train small neural networks, achieving estimations of similar quality. In [11], certain distribution properties of the problem were included in similar models in order to achieve more precise estimations in instances where points are not linearly distributed.

For estimating an optimum TSP tour length, a simple feed-forward neural networks with a single hidden layer and ReLU activation was employed. The input features were derived from [10]. The network estimates the minimum TSP tour length given a set of support points that lie equidistantly on that tour and operates on the features listed in Table 1.

A (fully connected) neural network is a sequence of linear mappings with nonlinear functions applied after each stage. Each occurrence of such a linearity/nonlinearity combination is called a layer. The computation in the k-th layer can thereby be expressed as follows:

$$x^{(k)} = f^{(k)}(x^{(k-1)\top} \cdot W^{(k)} + b^{(k)})$$

For the linear mapping, W is called a weight matrix and b is a bias term. The input is $x^{(0)}$, the output is $x^{(K)}$ with K being the number of layers. Theoretically, the complexity of the mapping a neural network can compute increases with the amount of layers it possesses as well as the layer size itself [12].

Table 1 List of TSP features that were used to estimate the optimal tour length where predetermined support points lie equidistantly on that tour

Feature	Definition
$\sqrt{N_w}$	Square root of the number of points that are not support points
$\sqrt{A_w}$	Square root of the area of the smallest rectangle that covers all points that are not support points
R_w	Ratio of height to width or width to height of A_w (such that $0 < R_w < 1$)
$\sqrt{N_s}$	Square root of the number of support points
$\sqrt{A_s}$	Square root of the area of the smallest rectangle that covers all support points
R_s	Ratio of height to width or width to height of A_s (such that $0 < R_s < 1$)
D_s	Average straight line distance to/from the support points
D_{in}	Average straight line distance from the start point
D_{out}	Average straight line distance to the end point

The network possesses 210 trainable parameters and was trained on 7200 training samples with a learning rate of 10^{-2}. The optimum tours required as a baseline for error backpropagation were computed using a variation of the Held-Karp algorithm [13]. As a loss function, the $L1$ loss was used.

Training samples were randomly generated on a 10×10 grid with instance sizes ranging from 6 to 25 and 3 to 5 support points in between. Generating samples this way however creates a heavy bias toward instances with average length, because the optimum tour lengths adhere to a normal distribution. This is suboptimal for learning, since the interesting cases of short tours are especially underrepresented. We therefore created the training set such that the tour lengths were linearly distributed by discarding samples the length of which was already sufficiently represented in the set.

We ran the network on a test set consisting of 3150 samples. The mean average precision error (MAPE) was 0.135. 45.2% of all estimations had less than 10% error, 78.3% had less than 20% error. The exact distribution of relative errors in shown in Fig. 4. In the experiments in Sect. 3, we opted to use error margin factors ranging from 0.95 to 0.8 in order to compensate up to 20% estimation error.

Figure 5 shows the distribution of optimum tour lengths of the test set where the relative error is above 20%. As one can see, short tours tend to be more difficult to estimate precisely than longer tours. Alas, shorter tours are of specific interest for us, because for our scheme to be efficient we need to resolve differences between the shortest tours as precisely as possible. Optimal TSP tour lengths for instances of size > 1000 can be estimated significantly better ($1-2\%$ relative error) at a constant network size [10, 11]. We therefore expect the algorithm to become more efficient as problem sizes grow since the error margin factor can be very close to 1 and the minimum length tour can thus be chosen with more confidence.

For this work we stuck with small problems of size 13–21, because larger problems would require a tour length estimator capable of handling larger instances. Training such an estimator is conditional on being able to solve large TSP/MHP instances with fixpoints efficiently. Such a solver exists only for the standard TSP formulation at the

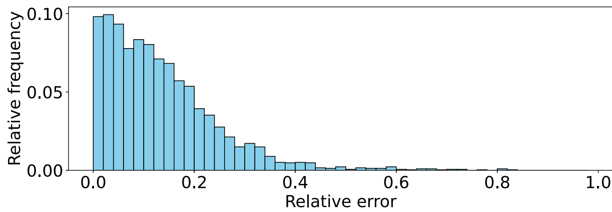

Fig. 4 Distribution of relative errors of the optimum tour length estimations returned by our neural network on a test set containing 3150 samples

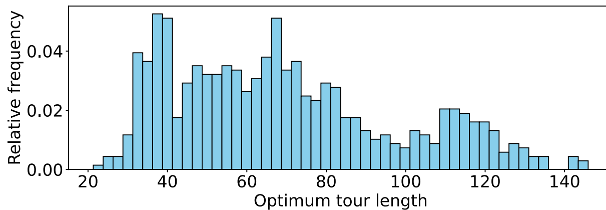

Fig. 5 Distribution of the optimum tour lengths for instances in the test set where the estimation by the neural network was at least 20% off

time of writing – the implementation of a custom solver is out of scope of this paper and beyond that not required for the analysis in Sect. 3. In turn this means that the only prerequisite to applying our scheme to larger instances is the realization of an estimator trained on larger instances.

2.3 Solving TSP with QAOA

For solving the subproblems generated by the A^* scheme, we used the quantum alternating operator Ansatz; hence, we encoded hard problem restrictions in the mixer. The TSP setup from [9] was adapted to fit the MHP requirements.

The mixer is constructed from 4-qubit operations, where the points in time of cities consecutive in a tour are swapped, for example: If city n is visited in the first step and city m in the second, the application of one such gate would have city m be visited first and city n second. In contrast, a standard QAOA mixer would not consider problem constraints and transfer any (possibly invalid) solution encoding to some different one.

This 4-qubit swap operation has been implemented in gates as depicted in Fig. 6 and was arranged as a so-called color-parity ordering swap mixer as described in [9]. The gates for the phase separator were arranged in an analogous fashion, resulting in minimal circuit depth for this QAOA setup. Small alterations to the phase separator were made such that the MHP problem formulation is adhered to. The Qiskit source code of our implementation can be found at[1].

As an optimizer the constrained optimization by linear approximation optimizer (COBYLA) was used and the circuit ran on a noiseless simulator. In 1000 randomly generated instances of size 4, the approximation was on average 8.07% worse than the optimum solution. The complete distribution of relative approximation errors is depicted in Fig. 7. 68.4% of all samples were solved optimally; hence, very few heavy

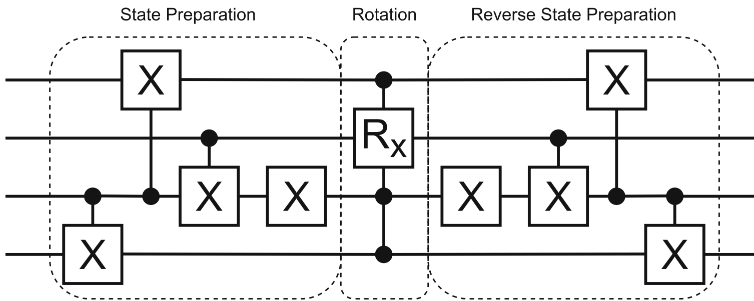

Fig. 6 Gate implementation of a 4-qubit swap operation that parametrically swaps the states $|0110\rangle$ and $|1001\rangle$. The operation has been verified through the desired unitary matrix representation

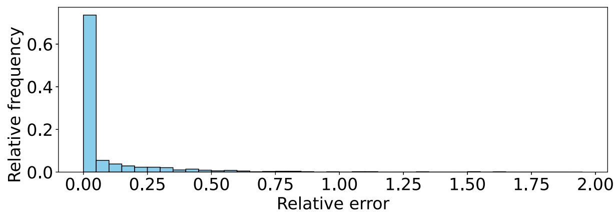

Fig. 7 Distribution of the relative approximation error on 1000 randomly generated TSP instances of size 4, solved with QAOA on a noiseless simulator and using COBYLA as the optimizer. 68.4% of all results were without error

outliers (for example 1 instance where the approximation was 150% times larger than the optimum) skew the average approximation quality.

3 Experiments

In this chapter we illustrate the behavior of the algorithm under different conditions such as optimal or approximate subsolutions, varying problem sizes and different error margin factors. As the estimation of a TSP/MHP tour length is difficult for small instances, we chose to implement ESTIMATE_MHP_SUPPORT as described in 2.2 and let ESTIMATE_MHP be determined by an optimal solution obtained via dynamic programming, since the neural network estimations for that problem proved to be too erratic for instances of size 4–6.

We first examine an example TSP instance with 17 points and the different solutions thereof shown in Fig. 8.

(a) shows the optimal solution of length 78.83 obtained with the Held-Karp algorithm [13].

(b) shows the application of our algorithm using CANDIDATES_STEP from Sect. 2.1, an error margin factor of 0.95, the tour length estimation network described in Sect. 2.2 and subsolutions that were solved optimally. The highlighted tour is therefore an approximation, since the tour length estimation is not admissible and many candidates in each step were pruned, albeit with length 79.01 only marginally longer than the optimum.

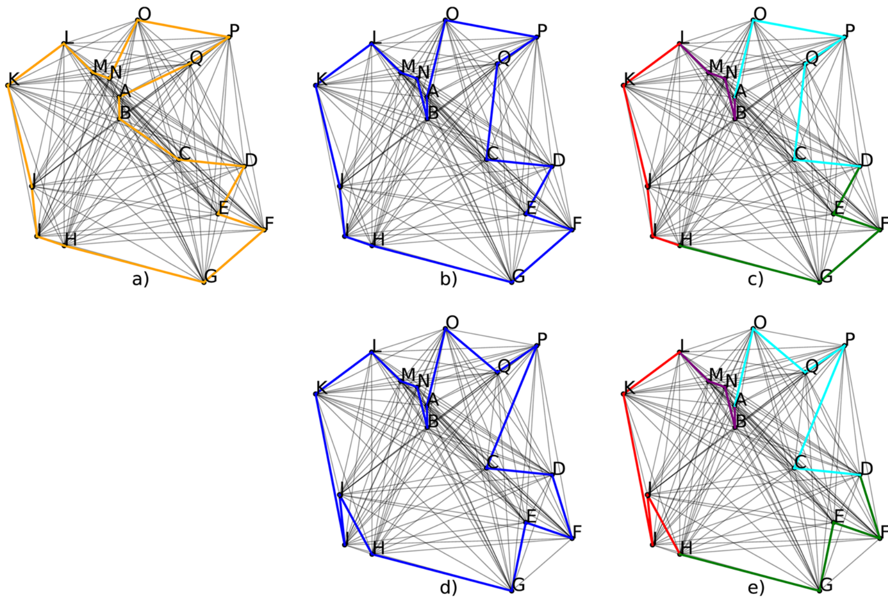

Fig. 8 Example TSP instance with different solutions marked. **a** is the optimal solution. **b** shows an approximation using our algorithm and optimal subsolutions. **c** highlights the individual optimal subsolutions. **d** shows an approximation using our algorithm and subsolutions obtained via QAOA. **e** highlights the individual QAOA subsolutions

(c) shows the subtours of that approximative solution. One can derive from the graph that the chosen support point sequence $A \rightarrow L \rightarrow H \rightarrow D$ is different from the optimal support point sequence $A \rightarrow E \rightarrow I \rightarrow M$.

(d) shows the application of our algorithm with identical parameters as for (b), the subtours were solved using QAOA though. The tour of length 86.42 is noticeably longer than the optimum.

(e) shows that the procedure using QAOA found the same support point sequence as the optimal subtour execution, which is not guaranteed for approximative subsolutions. It is evident however that the tour between support points is often not optimal, resulting in a longer overall tour.

Moving on, we illustrate how the algorithm probably behaves in general, not to give exact (expected) measurements, since we are dealing with toy problems by necessity at the moment. For all following experiments we solve the generated subproblems classically through dynamic programming, since simulating QAOA is runtime intensive.

Figures 9 and 10 show the effect that the error margin factor has on the approximation error and the number of subinstances that have to be solved for problems of varying size. Problems sizes n are one of {10, 13, 16, 19} and all of them are solved in 3 steps, i.e., $k = \frac{n-1}{3}$. We randomly generated 1000 instances for each size.

In Fig. 9, the increase of error as the error margin factor increases for $n = 10$ **a** to $n = 19$ **d** is shown. The mean and median of the relative errors are usually close together and only grow closer as the problem size increases, suggesting a favorable

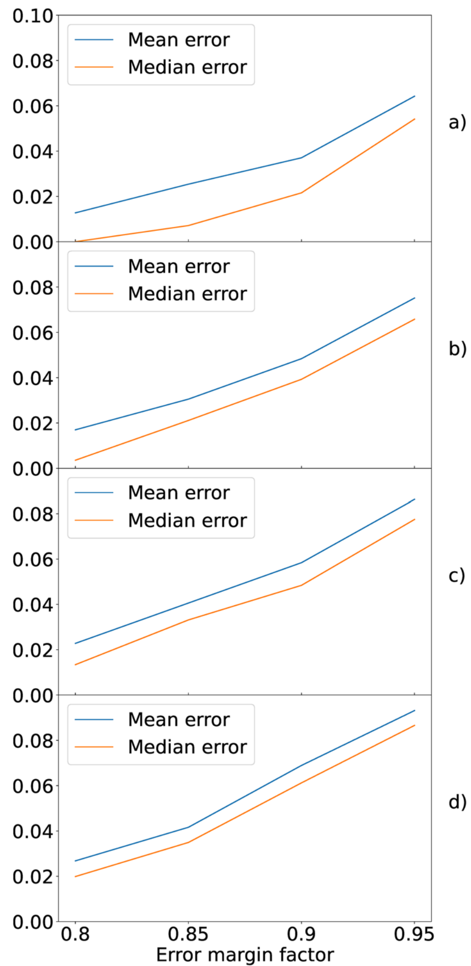

Fig. 9 Mean and median relative errors dependent on error margin factor of 1000 instances with $n = 10, k = 3$ **a**, $n = 13, k = 4$ **b**, $n = 16, k = 5$ **c**, $n = 19, k = 6$ **d**

error distribution and higher estimation uncertainty for small problems. The latter observation coincides with our findings in Sect. 2.2. The relative error grows linearly with the error margin factor.

Figure 10 depicts the decrease of the number of solved subinstances as the error margin factor increases for $n = 10$ **a** to $n = 19$ **d**. This number appears to decrease exponentially as the error margin factor increases. The mean deviates noticeably from the median, indicating a small percentage of instances where the number of subproblems grew disproportionally large. We observed this behavior in the following experiments as well and is probably caused by the imperfect tour length estimator. The algorithm should nevertheless behave favorably in most instances.

Strikingly obvious from both figures is the observation that the number of solved subproblems drops rapidly for higher error margin factors while the introduced error grows slowly. This suggests that high error margin factors, even with suboptimal estimators, should be beneficial in a runtime versus error consideration.

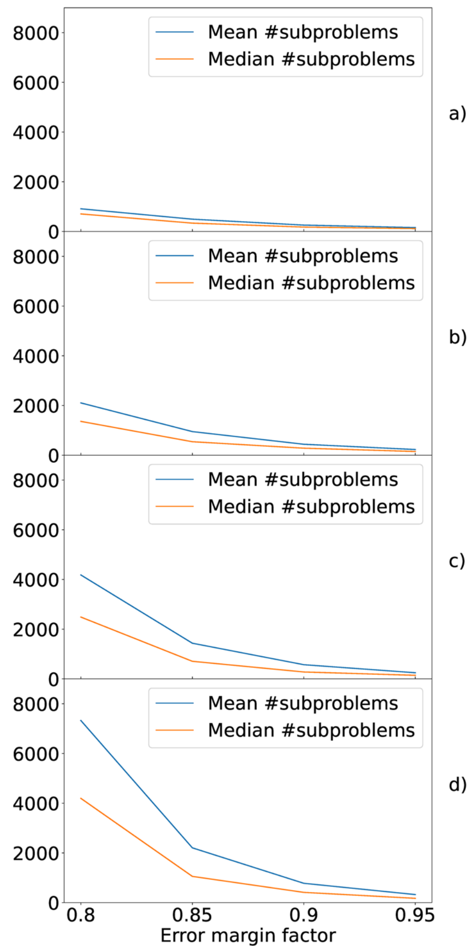

Fig. 10 Mean and median number of subproblems solved dependent on error margin factor of 1000 instances with $n = 10, k = 3$ **a**, $n = 13, k = 4$ **b**, $n = 16, k = 5$ **c**, $n = 19, k = 6$ **d**

In Table 2, the values for error margin factor $e = 0.95$ from the figures above are listed. One can see that for a high enough error margin factor the number of solved subinstances grows slowly compared to $e = 0.8$. As the quality of tour length estimations increases alongside the instance size, allowing e to be chosen very closely to 1 without violating the admissibility in too many cases, the number of solved subproblems should remain manageable as long as the number of steps is constant.

As can be discerned from Figs. 9 and 10 as well as Table 2, the relative error distinctly grows from (a) to (d). One way of explaining that growth is to argue that from smaller to larger problems, the fraction of subproblems that are solved compared to the number of subproblems that theoretically exist, becomes smaller. The overall problem is therefore "less explored" for larger n. This idea fits well with how smaller error margin factors result in more subproblems being solved, in turn causing the relative error to decrease. The implication here is that the choice of candidates in each step in CANDIDATES_STEP is crucial for the overall performance of the algorithm.

Table 2 Mean and median values for relative approximation errors and number of subinstances of 1000 runs with $n \in \{10, 13, 16, 19\}$, $e = 0.95$, $k = \frac{n-1}{3}$

n	Relative approximation error		Number of subinstances	
	Mean	Median	Mean	Median
10	0.064	0.054	156.5	117.0
13	0.075	0.066	228.1	149.0
16	0.086	0.077	247.83	145.0
19	0.093	0.087	324.19	173.5

Table 3 Mean and median values for relative approximation errors and number of subinstances of 1000 runs with $n = 13$, $e = 0.95$, $k \in \{3, 4, 6\}$

k	Relative approximation error		Number of subinstances	
	Mean	Median	$Mean$	$Median$
3	0.13	0.11	550.26	255.0
4	0.075	0.066	228.1	149.0
6	0.036	0.023	66.38	45.0

Table 4 Mean and median values for relative approximation errors and number of subinstances of 1000 runs with $n \in \{9, 13, 17, 21\}$, $e = 0.95$, $k = 4$

k	Relative approximation error		Number of subinstances	
	Mean	Median	$Mean$	$Median$
9	0.022	0.0	48.36	36.0
13	0.075	0.066	228.1	149.0
17	0.13	0.12	1068.81	543.0
21	0.18	0.17	8698.97	3260.0

Table 3 lists the relative errors and number of subinstances solved for $n = 13$, $e = 0.95$ and $k \in \{3, 4, 6\}$. Both items become smaller as k increases, which aligns with our previous findings. For a higher ratio of $\frac{k}{n}$ the fraction of solved subinstances to theoretically existing ones is higher, lowering the overall error. It is therefore very beneficial to let k be as large as is possible on a given quantum machine.

In Table 4 relative errors and number of solved subinstances for $e = 0.95$, $k = 4$ and $n \in \{9, 13, 17, 21\}$ is shown. The number of solved subinstances especially grows very rapidly. This is probably caused by the lack of initial pruning – the number of initial nodes for $n = 9$ is $\binom{8}{2}$, decidedly smaller than $\binom{20}{5}$ for $n = 21$. Node generation heuristics as discussed in 2.1 have a major impact on runtime and approximation quality, making careful construction of those an essential part of achieving practically relevant performance.

Overall we expect the scheme to perform well on larger instances where tour length estimations are more reliable. There is evident need for efficient heuristics that reduce

the number of generated subproblems. Using a high enough error margin factor should then result in quick approximations with small errors, since those are dependent on said heuristics and tour length estimations only.

4 Discussion

We presented an algorithm based on A^* that decomposes large instances of the TSP into small subproblems of a fixed size that can be individually solved or approximated on a quantum computer and subsequently recombined into a larger solution to the initial problem. To support this method, we provided several heuristics that drastically reduce the size of the search tree. We also adapted existing methods for TSP tour length estimation to our use case where the estimation needs to consider equidistant support points on the returned route. Additionally, we constructed and implemented a quantum circuit, including a problem specific mixer realization, that can approximate a TSP solution using QAOA. In experiments, we analyzed the behavior of the algorithm and checked whether that behavior matches intuitive expectations.

We found that all techniques to reduce runtime have tangible impact on the quality of the approximations. While the approach is very flexible, those techniques have to be carefully developed in order to obtain competitive solutions.

While the best case runtime of our algorithm with heuristic optimizations in place is comparable to those of classical approximation algorithms, the average case performance is still to be verified for larger instances. Experiments of that nature require tour length estimators that don't exist yet, since their training requires exact solvers that solve TSP/MHP problems with regard to a set of support points.

In the same vein, the quality of the approximations is not only dependent on tour length estimations and heuristics but also on the quality of the solution the quantum machine returns. While QAOA can theoretically compute approximations that are arbitrarily close to the optimum, this is only a theoretical guarantee that can heavily impact runtime [3]. We therefore considered the option to insert any preferred solver instead of QAOA essential for the utility of the scheme.

Putting this in perspective, we found a generic approach to partition a combinatorial optimization problem into smaller instances, that can be solved on a quantum computer by any chosen algorithm. The implication is that solving those problems can utilize the advantages that quantum computing provides, more effectively and at an earlier point in time as quantum computers are being developed.

Funding Open Access funding enabled and organized by Projekt DEAL. This work has been supported by the Ministry of Economic Affairs Baden-Württemberg in the project SEQUOIA under project number 036-840012.

Data Availability The source code for a complete implementation of this work can be found at https://github.com/ekut-es/decomposition-for-quantum.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Leymann, F., Barzen, J.: The bitter truth about gate-based quantum algorithms in the NISQ era. Quantum Sci. Technol. **5**(4), 044007 (2020). https://doi.org/10.1088/2058-9565/abae7d
2. Farhi, E., Goldstone, J., Gutmann, S., Lapan, J., Lundgren, A., Preda, D.: A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem. Science **292**(5516), 472–475 (2001). https://doi.org/10.1126/science.1057726
3. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm. arXiv (2014). https://doi.org/10.48550/ARXIV.1411.4028
4. Jünger, M., Reinelt, G., Rinaldi, G.: The traveling salesman problem. Preprint, Universität zu Köln (1995). https://kups.ub.uni-koeln.de/54671/
5. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River (2009)
6. Raphael, C.: Coarse-to-fine dynamic programming. IEEE Trans. Pattern Anal. Mach. Intell. **23**(12), 1379–1390 (2001). https://doi.org/10.1109/34.977562
7. Montiel, O., Díaz Delgadillo, F.: Reducing the size of combinatorial optimization problems using the operator vaccine by fuzzy selector with adaptive heuristics. Math. Probl. Eng. (2015). https://doi.org/10.1155/2015/713043
8. Pelofske, E., Hahn, G., Djidjev, H.: Decomposition algorithms for solving np-hard problems on a quantum annealer. J. Signal Process. Syst. **93**(4), 405–420 (2021). https://doi.org/10.1007/s11265-020-01550-1
9. Hadfield, S., Wang, Z., O'Gorman, B., Rieffel, E.G., Venturelli, D., Biswas, R.: From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. Algorithms (2019). https://doi.org/10.3390/a12020034
10. Kwon, O., Golden, B., Wasil, E.: Estimating the length of the optimal TSP tour: an empirical study using regression and neural networks. Comput. Oper. Res. **22**(10), 1039–1046 (1995). https://doi.org/10.1016/0305-0548(94)00093-N
11. Çavdar, B., Sokol, J.: A distribution-free tsp tour length estimation model for random graphs. Eur. J. Oper. Res. **243**(2), 588–598 (2015). https://doi.org/10.1016/j.ejor.2014.12.020
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org
13. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. J. Soc. Ind. Appl. Math. **10**(1), 196–210 (1962). https://doi.org/10.1137/0110015

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.