DISPLACEMENT ACTIVITY: SOLVING THE SHORTEST COMMON SUPERSTRING PROBLEM VIA DEEP REINFORCEMENT LEARNING

A PREPRINT

Vinay Ayyappan
Department of Biomedical Engineering
Johns Hopkins University
Baltimore, MD 21218
vayyapp1@jhu.edu

Joanna Y. Guo
Department of Biomedical Engineering
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
joannaguo@jhu.edu

Ronan F. L. Perry
Department of Biomedical Engineering
Department of Applied Mathematics and Statistics
Johns Hopkins University
Baltimore, MD 21218
rperry27@jhu.edu

Hadley F. VanRenterghem
Department of Biomedical Engineering
Johns Hopkins University
Baltimore, MD 21218
hvanren1@jhu.edu

June 18, 2019

ABSTRACT

Hand-crafted heuristics for solving NP-hard optimization problems are fast and often effective; yet they lack theoretical guarantees and often fail to generalize across problem instances. Deep architectures thus bear promise for automated discovery and design of better heuristics that garner high approximation ratios. The shortest common superstring problem (SCSP) is a fundamental problem in computer science with applications in DNA sequence analysis, data compression, and text processing that seeks to find the shortest superstring that contains a set of input strings. The problem is NP-hard; thus the development of efficient algorithms for solving SCSP is of significant interest. We present a deep learning-driven approach for identifying close-approximated solutions for the SCSP that combine reinforcement learning with embedding. An evaluation function learned using a set of SCSP problem instances will be used to drive a greedy algorithm for constructing solutions.

***K*eywords** Shortest Common Superstring · genomics · reinforcement learning · deep learning

1 Introduction

The shortest common superstring problem (SCSP) seeks to find the shortest string that contains as substrings each among a set of input strings [1]. The SCSP arises in several applications including data compression, text processing, and especially DNA sequencing, where a solution must be formulated to align large numbers of randomly-sequenced DNA fragments. Thus, this particular application of the SCSP seeks optimal solutions over an alphabet of four letters. The SCSP is known to be NP-hard [2], but it can be modeled as the asymmetric Traveling Salesman Problem (ATSP) [3], for which efficient approximation algorithms exist due to Tarhio and Ukkonen [4], and Mucha [5]. In the ATSP, we are given a directed graph $G = (V, E)$ with weighted edges, and seek to find a Hamiltonian Circuit of minimum global cost [6]. This optimal tour cost provides a lower bound for the optimum of the shortest superstring [7]. Tarhio and Ukkonen propose a 312-approximation algorithm that greedily joins together pairs of strings of longest prefix-suffix overlap until a single string, the approximated shortest substring, remains [4]. While Ukkonen demonstrated an

$O(n)$-time implementation of this greedy algorithm in 1990 [8], research has since focused on identifying algorithms with improved space usage [9] and improved approximation ratios [10]. Work by Mucha provides the algorithm with the best established theoretical approximation ratio, of $2\frac{11}{23}$ [5].

These heuristics for solving NP-hard problems in discrete optimization, such as SCSP, are generally fast, and are often quite effective. Even so, hand-crafted heuristics for solving optimization problems often require significant modifications following slight changes to the problem statement. Recent work by Dai, Prates, and Caldwell has sought to exploit similarity among instances of such problems to develop better heuristics for problem solving [11, 12, 13]. Their use of deep architectures for the same has resulted in the discovery of novel algorithms that yield improved approximation ratios in reasonable running time for problem instances such as the Traveling Salesman Problem (TSP), Minimum Vertex Cover (MVC), and Maximum Cut (MAXCUT). This bears promise for similar approaches that harness deep learning techniques to generate close-approximated solutions for other problems, particularly when more tailored approaches are used that consider intrinsic problem properties. Here, we propose to employ an approach to solving the SCSP that combines reinforcement learning and embedding. An evaluation function learned using a set of problem instances will be used to drive a greedy algorithm for constructing solutions.

2 Background

2.1 The Shortest Common Superstring Problem

Given a set of strings $S = \{s_1, s_2, ..., s_n\}$, the SCSP seeks to find the shortest superstring that contains each substring among S. Modeling the SCSP as an ATSP, we may construct a directed graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ with each vertex v_i corresponding to string s_i. Asymmetry arises from the fact that superstrings can comprise pairings of strings s_i and s_j joined suffix to prefix, or prefix to suffix, with the levels of overlap associated with each case not necessarily being equal. There are three cases of string pairing that any solution to SCSP will need to handle. First, given two strings with an overlap of m characters, we have that the length of the corresponding superstring is $|s_i| + |s_j| - m$. If there is no overlap between s_i and s_j, the two may simply be concatenated. Moreover, in the event that one of s_i or s_j is a proper substring of the other, then any superstring containing the longer string also contains the other; thus the shorter string may be discarded. Any edge can correspond to a string-string pairing; thus G is a complete graph, and the weight of the directed edge $e_{ij} \in E$ from vertex v_i to vertex v_j is equal to the length increase in s_i resulting from prepending s_i to s_j. These edge-weights can be quickly computed during preprocessing by finding the maximum prefix-suffix overlap between each ordered pairing of strings. During this preprocessing stage, vertices corresponding to any string that is a proper substrings of another can be deleted; vertices corresponding to repeated strings can be deleted until only one copy remains. A solution to the ATSP on G is a solution to the SCSP on S.

2.2 Genome Sequencing

The SCSP commonly arises in the arena of genomics, where large genomes comprising vast sequences of nucleotides (in this case, an alphabet comprising four letters, A, C, T, G) take the form of strings comprising on the order of 10^9 characters [14] [15]. While technology does not exist as-yet to read these genomes in their entirety, a number of genome sequencing strategies exist, such as “shotgun sequencing” that enable generation of short fragments (substrings) of a given genome, referred to as “reads,” that enable subsequent reconstruction of the original genome superstring. Thus, the problem of “re-assembling” these genomes can be formulated as an SCSP, where the original genome is the shortest common superstring comprised of each reads. A graphical representation of this problem would therefore possess, as its vertices, each read, and as its edges, the prefix-suffix overlaps among pairs of reads. In reality, genome sequencing makes occasional errors that result in imperfect overlaps and incomplete coverage among reads [16]; however, the SCSP formulation of the assembly problem, which largely ignores this noise, is nevertheless useful for informing approaches for rapid and accurate reconstruction of genomes, especially given new methods that use the topology of graph representations of substring-substring relationships to correct such read errors.

Solvers of the SCSP often assume reads of fixed-length; while this simplifies the problem greatly, it is not always accurate. In particular, while Next Generation Sequencing and Illumina sequencing technologies can be used to generate fixed-length substrings, other technologies, such as 454-sequencing, do not [16]. While fixed-length substrings (often referred to as k-mers) can be generated from the variable-length reads, this method often results in a loss of information from reads when repeated sequences extend longer than the chosen fixed length. Moreover, this choice of length entails several trade-offs: longer substrings require fewer vertices, but greater space to store each substring; conversely, shorter substrings risk losses of read information on repeated sequences, as described above [16]. It is therefore useful to consider formulations of the assembly problem that consider variable-length reads.

2.3 Prior Approaches

Early approaches for solving SCSP were directed toward efficient solvers with low operating-time. These approaches, such as that developed by Tarhio and Ukkonen, have manifested as greedy approximation algorithms that operate by first computing the maximum pairwise overlaps among a set of substrings, pre-processing by removing any substring that happens to be a proper substring of another, and then greedily forming a superstring by adding to the solution substrings that have the greatest available overlap with the growing superstring. This approach garners an approximate solution in $O(n)$ time [8].

Since then, efforts have become directed toward solutions with better approximation ratios. To-date, the best among these was devised by Mucha, with a theoretical approximation ratio of $2\frac{11}{23}$ [5]. This algorithm operates by first constructing "representative strings" that contain each substring in a cycle of a minimum cycle cover of a graph composed of pairwise "prefixes," i.e. the portions of any given substring that do not overlap with another substring. Each representative substring is subsequently reduced to a Max-ATSP Path problem, where the objective becomes to construct the maximum-length tour over the overlap graph corresponding to each representative string. This can then be the point of intervention for a greedy algorithm, formulated as above. It is worth noting, however, that this greedy approach, which only selects the best node immediately-available, is myopic, i.e. it does not necessarily consider the best sequence of actions to take over the long term. As a result, the greedy approach may be very effective in for solving problems involving random datasets comprising fixed-length substrings, but not those involving variable-length substrings or strings with many repeated fragments. In these cases, some substrings may happen to become proper substrings of the growing solution; thus, the best action over the long term may not necessarily be to add to the growing solution that substring with the greatest-length overlap with the solution.

In contexts involving genomics, genetic algorithms, such as Deposition and Reduction, Enhanced Beam Search, and Chemical Reaction optimization have been especially popular as approaches used in-place of greedy algorithms. In particular, in 2006, Ning developed a Deposition and Reduction algorithm, which works by constructing a template pool of sequences using a "Look Ahead Sum-Height" algorithm that inserts into the growing superstring the substring that will result in the greatest available increase in superstring length, evaluating and the effect of each insertion over several steps ahead [17]. More recently, Mousavi developed an Enhanced Beam Search algorithm that employs a probabilistic heuristic to prune the search space as solutions are constructed [18]. Lastly, Saifullah developed a Chemical Reaction Optimization method that combines global and local search properties in an effort to identify SCSP solutions [19]. These latter approaches are highly experimental and less well-characterized. They are thus greatly in need of further evaluation.

3 Methodology

3.1 Method Overview

We assume that instances of the SCSP lie on some distribution of such problem instances; thus, it is possible to generate trained heuristic methods for solving the SCSP. Treating a graph representation G (**Fig. 1A-B**) as one among a distribution D of such representations of the SCSP, a heuristic is learned that generalizes over a set of instances from D [11]. This heuristic approach adds vertices x to a partial solution X until a complete solution is achieved. As formulated by Dai, a helper function $h(X)$ maps X to a combinatorial structure that satisfies the given SCSP instance and whose quality with respect to graph G is defined by cost function c. Vertices are successively selected and added to X by maximizing an evaluation function Q, whose parameters are learned via deep learning.

The evaluation function is to be trained using end-to-end reinforcement learning, wherein all parameters are learned simultaneously [11]. Each partial solution X represents a state of the system; thus, each action of adding a new vertex to the partial solution leads to a new state. Such transitions between states are rewarded by changes in the cost function c. Thus the cumulative reward is the cost of the current solution. This is the basis for a greedy policy that maximizes Q at the action stage.

The evaluation function Q must not only summarize the partial solution X and its mapping with respect to the input graph G, but also identify the value of new nodes to be added to the graph. To represent our partial solution as it relates to the graph structure, we first define a binary decision variable x_v that corresponds to each node v in G, so that $x_v = 1$ if $v \in X$. In this manner, we essentially "tag" nodes within the partial solution. Next, to train Q to account for this partial solution X and to evaluate the quality of new nodes to be added, a non-graphical representation of vertices must be used that also preserves the relationships as described above among vertices. While Dai demonstrated an effective embedding using an algorithm that uses graphical model inference on a Markov random field to learn a latent k-dimensional vector μ_i for each vertex v_i in an undirected graph, these are, by definition, undirected, and thus invalid

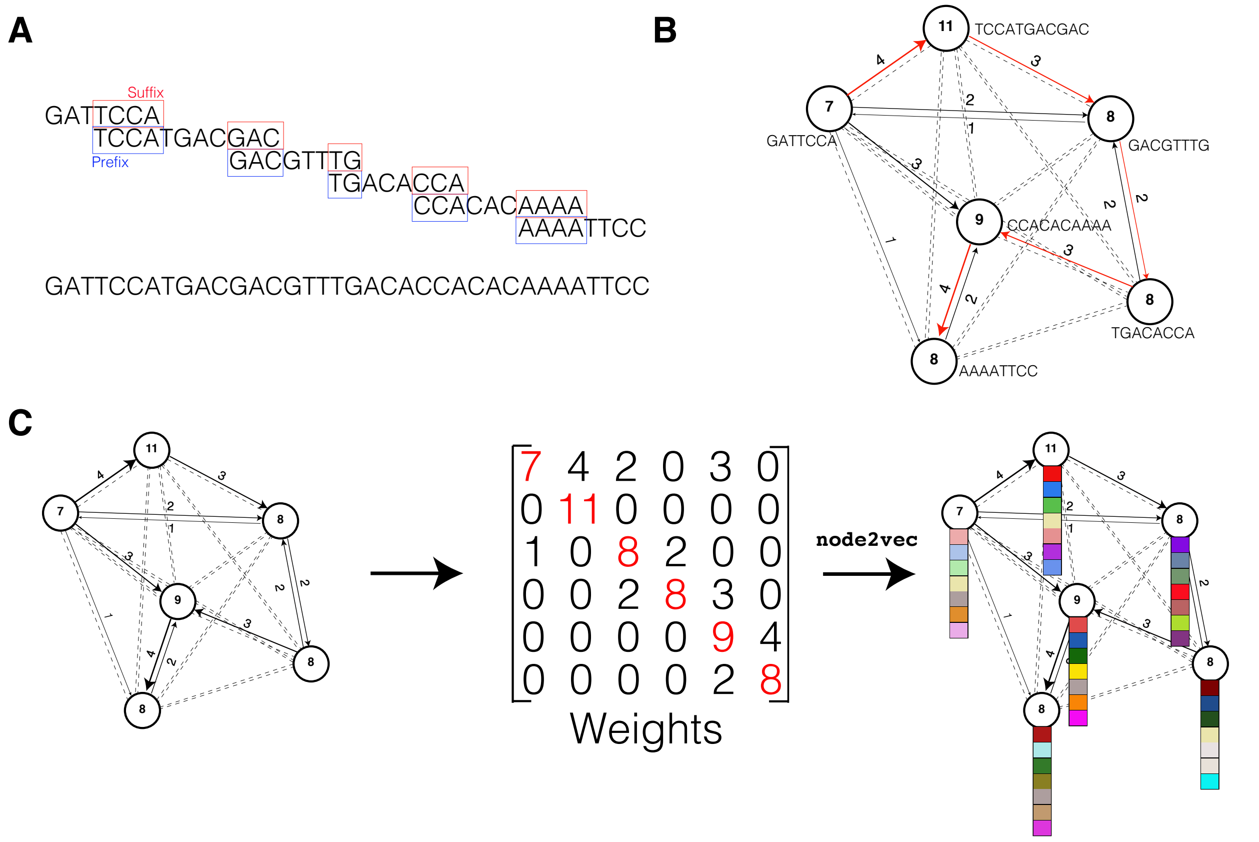

Figure 1: Overview of SCSP and our embedding workflow. **(A)** Example of substring alignment and generation of superstrings. **(B)** Example of a graph representing the strings aligned in panel (A). **(C)** Schematic demonstrating the embedding workflow employed. A weights matrix is formed using the respective overlaps among substring-substring pairs. Since each node is "labeled" according to its length (an intrinsic cost), this information is stored along the diagonal of the weights matrix. A k-dimensional embedding is learned for each node via `node2vec`.

for our ATSP formulation. We therefore employ the graph-embedding network `node2vec` to featurize vertices in our graph [20].

3.2 Graph Embedding

Any representation of the graphs corresponding to an SCSP instance must encode not only the "edge-specific cost" of overlaps between pairs of strings but also the "intrinsic cost" of each vertex. As such, we first generate a matrix M that captures this information, such that each entry m_{ij} corresponds to the length of overlap between the suffix of substring s_i and the prefix of substring s_j. Diagonal elements of M encode the lengths of each substring s_i [21]. To encode values associated with each edge e_{ij}, the overlaps among each possible pairing of vertices v_i and v_j, we generate a k-dimensional embedding of the nodes in each graph using, `node2vec`, a two-layer neural network that maps each node to a low-dimensional space (**Fig. 1C**) [20]. This mapping maximizes the likelihood of preserving each node's neighborhood. After constructing the initial embedding v for each node $v \in V$, embeddings are synchronously updated during training by a neural network, as performed by Dai:

$$\mu_v^{(t+1)} = relu(\theta_1 x_v + \theta_2 \sum_{u \in N(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in N(v)} relu(\theta_4 m_{v,u}))$$

Here, $N(v)$ corresponds to the set of neighbors of node u in G. θ_1, $\theta_4 \in \mathbb{R}^k$ and θ_2, $\theta_3 \in \mathbb{R}^{k \times k}$ are model parameters, and $relu$ is the rectified linear unit applied element-by-element to its inputs, i.e. $relu(x) = \max(0, x)$. As formulated by Dai, summing over neighbors enables aggregation of neighborhood information. Updated embeddings generated in this regard after some choice of T iterations will be used to define the evaluation function $Q(h(X), v; \theta)$.

The embeddings μ_v^T for each node $v \in V$ are subsequently pooled over the entire graph so that $Q(h(X), v; \theta) = \theta_5^T relu([\theta_6^T \sum_{u \in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}])$. Here, $\theta_5 \in \mathbb{R}^{2k}$ and θ_6, $\theta_7 \in \mathbb{R}^{k \times k}$. The parameters $\{\theta_i\}_i^7$ are learned via end-to-end reinforcement learning.

Notably, the above formulation on its own only considers movement along the graph in the forward direction, i.e. the construction of solutions by appending substrings to the suffix of the growing superstring. In order to also account for additions to the solution formed by pre-pending new substrings to the partial solution, we consider a "reverse embedding" update, formed by evaluating the Q function after providing the transpose of the matrix M, i.e. we update our embedding so that the argument of the second $relu$ in the embedding update function provided above is $\theta_4 m_{u,v}$. For clarity, we will subsequently refer to the Q-function learned from the "forward embedding" as Q_f and that formed from the "reverse embedding" as Q_r. Subsequently, as solutions are constructed, the best direction of movement will be chosen, as determined by reinforcement learning, and the corresponding Q-function will be referred to as $\hat{Q}$. This function will take the maximum at each time-step t of Q_f and Q_r.

3.3 Reinforcement Learning

Prior work, in particular, by Dai [11] and Mnih [22], utilizes reinforcement learning as a framework for learning parameters of the evaluation function $\hat{Q}(h(X), v; \theta)$. This work has provided a foundation for efforts to use deep reinforcement learning in applications including gameplaying [1]. Our framework similarly defines states, actions, and rewards as follows:

States: we define states X as sequences of nodes, tagged as above, on a graph G. We represent these nodes as vectors in k-dimensional space, where the final state X^* is unique to each given problem instance.

Transitions: transitions correspond to the identification of the most recently visited node $v \in G$. Operationally, this is defined by tagging this node in the growing partial solution X.

Actions: Each action is a node not included in the current state but selected for inclusion in the subsequent state. Each action, as is true of any node in our formulation, is represented according to its k-dimensional embedding.

Rewards: We define rewards $r(X, v)$ as the change in the cost function at state X following action v. Thus, the aggregate reward $R = \sum_{i=1}^{|X^*|} r(X_i, v_i)$, once the terminal state has been reached, corresponds with the objective function value at this terminal state X^*.

While end-to-end learning alone would result in a myopic strategy that only considers rewards at single-steps, we use n-step learning and Q-fitted learning in addition to reinforcement learning. At each step, a stochastic gradient-descent method is used to minimize the squared loss of the evaluation function $\hat{Q}$ with respect to some target y, set to be the maximum reward after n future steps. Q-fitted learning allows storage of partial solutions in some dataset $\mathcal{A}$ with storage capacity N during training and, via a technique known as experience replay, performance of stochastic gradient descent updates on randomly-drawn samples from $\mathcal{A}$. The dataset $\mathcal{A}$ stores partial solutions pooled over previous episodes such that after n steps (i.e. step $t + n$), a tuple, $(X_t, a_t, \sum_{i=1}^{n-1} r(X_{t+i}, a_{t+i}), X_{t+n})$, is added to $\mathcal{A}$. Further, as described above, we evaluate potential future partial solutions obtained by pre-pending or appending new vertices to the partial solution. While in theory, two separate networks, referred to as the Q network and the target network, can be used to respectively estimate the evaluation function Q and target y, our implementation uses the same uses identical architectures for the Q- and target networks.

Alone, the above approach would simply seek to minimize the squared loss of the Q-function with respect to the current solution X_t,

$$(y - \hat{Q}(h(X_t), v_t; \theta)^2$$

where $y = \sum_{i=1}^{n-1} r(X_{t+i}, v_{t+i}) + \gamma \max_{v'} \hat{Q}(h(X_{t+n}), v'; \theta)$ is a target formulated based upon an estimate of future rewards amassed after n-steps. The value $\gamma \in [0, 1]$ is a discount factor that dictates the relative importance of current and future rewards. Yet implementations of such Q-learning may risk learning unrealistically high action values due to their use of a maximization step over estimates of future actions. Thrun and Schwartz [23] give specific examples of game-playing and pathfinding applications where such overoptimism compromises resulting policies. To remedy this, we employ double Q-learning to decompose the "maximization" step into action selection and evaluation components [24]; i.e. rather than the selecting action v_i and calculating $\hat{Q}(h(X_{t+n}), v'; \theta))$ simultaneously, the target network evaluates solution quality, leaving selection of actions v_i to the Q network. Thus, the target assumes the following form:

[1]https://github.com/aurelienbibaut/DQN_MVC

$$y = \sum_{i=0}^{n-1} r(X_{t+i}, v_{t+i}) + \gamma\hat{Q}(h(X_{t+n}, \underset{v'}{\operatorname{argmax}}\,\hat{Q}(h(X_{t+n}), v'; \theta); \theta')$$

Algorithm 1: Deep Q-Learning

```
1  Initialize replay memory $\mathcal{A}$ with size $N$ Initialize evaluation function $Q$ with random weights
2  for episode 1, M do
3      Draw graph $G$ from its distribution
4      Initialize the state to empty $X_1 = ()$
5      for step $t = 1, T$ do
6          With probability $\epsilon$,select a random node $v_t$
7          otherwise, compute $v_{tf} = \operatorname{argmax}_v Q_f(h(X_t), v; \theta)$ and $v_{tr} = \operatorname{argmax}_v Q_r(h(X_t), v; \theta)$
8          Obtain $\hat{Q}(h(X_t), v; \theta) = max(Q_f, Q_r)$ and set $v_t = \operatorname{argmax}_v \hat{Q}(h(X_t), v; \theta)$
9          if $v_t$ == $v_{tr}$ then
10             Append $v_t$ to the partial solution: $X_{t+1} := (X_t, v_t)$
11         else
12             Prepend $v_t$ to the partial solution: $X_{t+1} := (v_t, X_t)$
13         if $t \geq n$ then
14             Add tuple $(X_{t-n}, v_{t-n}, \sum_{i=1}^{n-1} r(X_{t-n+i}, a_{t-n+i}), X_t)$ to $\mathcal{A}$
15             Sample a random batch $\mathcal{B}$ from $\mathcal{M}$
16             Update parameters $\{\theta_i\}_i^7$ over all $v_t$ for $\mathcal{B}$ via stochastic gradient descent
17 return parameters $\{\theta_i\}_i^7$
```

4 Implementation and Evaluation

4.1 Training, Testing, and Validation

We first generated data from a four letter alphabet (i.e. in the case of DNA). For each graph, we generated a uniformly-distributed random sequence comprising 100-1000 characters, and simulated shotgun sequencing in order to generate substrings of length 10-100. To do so, sequences were copied 10 times. Each copy was fragmented into segments to generate overlapping reads that originate at randomly-selected indices along each copy of the original sequence. These repeating substrings functioned as inputs to our algorithm. Copies were made to ensure that each position of the original string is represented on multiple fragments, such that when the fragments are overlapped in order, we are able to reconstruct the original string. The fragments of all the copies for each sequences were used as the nodes for our graphs. In this manner, we generated 800 graphs for training and 200 for testing.

Our first experiments involved evaluation of our methodology, here referred to as DDQ-SCS, on datasets comprising substrings of fixed-length, consistent with DNA sequencing technologies such as Illumina sequencing that sequence DNA in fixed-length reads. Next, validation for the fixed-length was performed using real datasets containing DNA sequencing information fragmented into 100 sequences of length 100, as provided by Saifullah [19]. Real DNA sequences, unlike random nucleotide sequences, such as those we provided during training, contain many long, patterned and repeated stretches, making it valuable to determine the ability of our method to generalize to such unseen cases.

All experiments were performed using a Google Compute Engine `n1-highmem-8` machine instance with 8 virtual cores and 52GB of memory. Parameters for learning were selected such that the network was trained on batches of size 32, and a replay capacity of 10,000 timesteps. An ϵ-greedy approach was employed during training such that a random action was performed (as opposed to the action evaluated as "best" per the heuristic being learned) with a probability p defined according to a decaying exponential with a time constant equal to 250 multiplied by the average number of actions to be performed (equivalent to the number of vertices) per graph. Our code is publicly-available. [2]

[2]https://github.com/rflperry/dl-scsp

4.2 Evaluation Criteria and Comparison of Solution Quality

We evaluated our solution quality on test instances using the approximation ratio, defined as $R(X, G) = \max(O(G)c(h(X)), c(h(X))O(G))$ for solution S to problem-instance G where $O(G)$ is the best-known solution value for problem-instance G, and $c(h(X))$ is the objective function value corresponding to solution X. Since we know the ground-truth length of the randomly-generated or real "genomes" from which our problem instances were procured, $c(h(X))$ corresponds to the length of these strings. However, in the specific case of real DNA sequences, the true length of the real shotgunned sequences is unknown; thus, we evaluate our method against the traditional greedy formulation. Moreover, since genetic algorithms have gained some recent interest as bases for solving SCSP instances [7, 17], we also compared our solutions to those provided by enhanced beam search, deposition and reduction, and chemical reaction optimization algorithms, implementing these methods for comparison by modifying existing open-source code.[3,4]

5 Results

5.1 Fixed-Length Reads

We first evaluated the performance of our method, DDQ-SCS, on a set of 1000 random sequences of length 100-1000 characters, comprising a four-letter alphabet (A, T, C, G) that was fragmented into fixed-length substrings containing 10-100 characters. The performance of DDQ-SCS in comparison to the traditional greedy algorithm, chemical reaction optimization (CRO), deposition and reduction (DR), enhanced beam search (IBS), and random insertion are shown in Table 1. As a control, we also include the performance of random insertions. The Greedy algorithm performed best, generating solutions with a mean approximation ration of 1.01, in reasonable run-time (in the case of 100-character genomes fragmented into 100 sequences of length 10 characters, the greedy algorithm generates solutions in 35ms on average). While DDQ-SCS lags behind the greedy approach, with an approximation ratio of 3.04, it does surpass some commonly used solvers, i.e. those that employ genetic algorithms or the greedy heuristic for solution construction, suggesting that the heuristics learned by DDQ-SCS do have potential to be used as effective solvers for the SCSP.

Table 1: Average Approximation Ratios on 200 Test Graphs - Fixed-Length Substrings

DDQ-SCS	Greedy	CRO	DR	IBS	Random
3.04	1.01	3.17	3.19	3.47	6.74

It is likely that improved parameter selection and more training data could allow DDQ-SCS to achieve even better results. As shown in **Fig. 2**, The training approximation ratio was consistently improving. Selection of learning parameters (e.g. learning rate, number of timesteps, etc.) was dictated in large part by limited computational resources and limited training data. As shown, by the time training had completed, the approximation ratio had not converged upon a particular value. Indeed, by the time training had been completed, 35 percent of the actions taken for training were random, suggesting that further improvement in testing results could be obtained had the model been trained on a greater number of graphs and employed a more finely-tuned learning schedule.

5.2 Real DNA Sequences

Finally, we sought to evaluate the ability of DDQ-SCS to generalize heuristics learned after training on fixed-length, randomly-generated sequences to real DNA sequences, which often comprise long stretches of repeated characters– a situation that can drastically complicate the given SCS problem-instance. We therefore compared the performance of DDQ-SCS with the traditional greedy algorithm on 5 test-instances of real genomic sequences (100 genome fragments of length 100). In this case, the true length of the "full genome" is unknown, however, the traditional greedy algorithm does perform better than DDQ-SCS, constructing superstrings with an average length of 9451.0. DDQ-SCS, on the other hand, generates superstrings with an average length of 9955.0, hardly better than simply concatenating substrings together. This suggests that the heuristic learned by DDQ-SCS fails to generalize properly to real genome sequences. Training on a larger cohort of graphs could potentially facilitate learning of better heuristics and avoid over-training.

[3]https://github.com/sgametrio/greedy-superstring
[4]https://github.com/khaledkucse/CRO_SCS

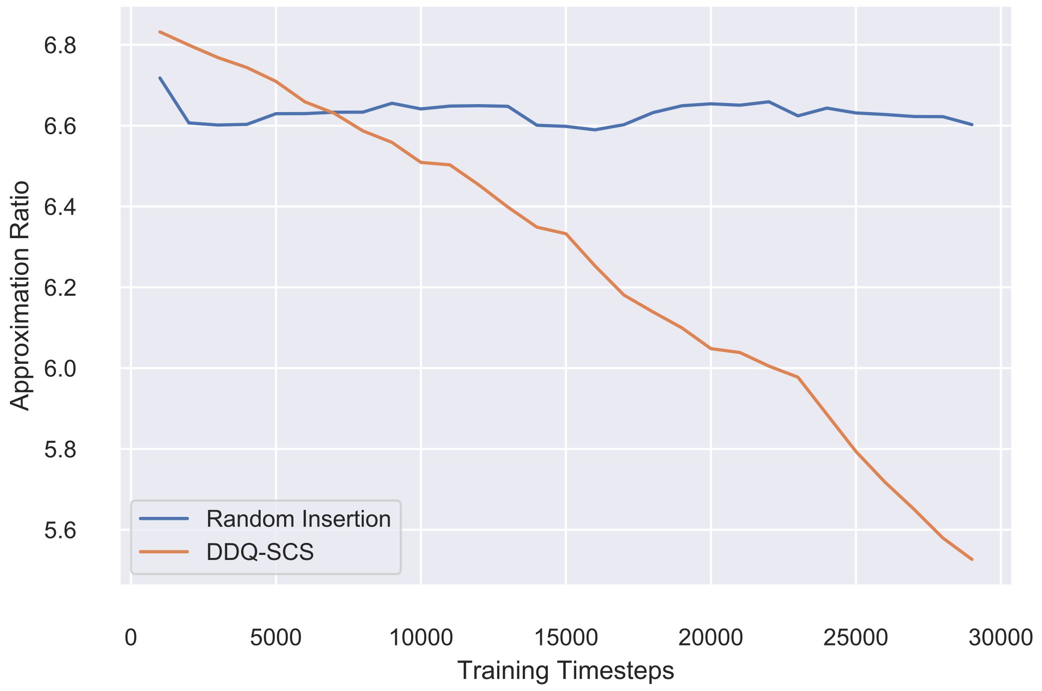

Figure 2: Training of DDQ-SCS showing improvement in approximation ratio over time for training of DDQ-SCS, in comparison with a random-insertion method. As shown, the approximation ratio appears to consistently improve.

6 Discussion

We present a deep learning-driven framework for learning heuristics to solve NP-hard combinatorial optimization problems on directed graphs. While our approach was formulated around the SCSP, our framework uses very little domain-knowledge specific to SCSP and can thus be easily extended to other problems, particularly those that can be formulated as an ATSP. Our approach primarily relies upon an embedding performed by deep neural networks, followed by end-to-end double Q-learning. This sets the stage for the learning of greedy-heuristics that can generalize across a distribution of problem-instances, thereby obviating the need for manually designing highly-tailored approaches.

Our results indicate that effective heuristics can be learned. For fixed-length reads, our results indicate an average solution approximation ratio of 3.04 that, while not as effective as the greedy approach, does perform better than some recently developed methods, including chemical reaction optimization, deposition and reduction, and enhanced beam-search. That said, any deficiencies in our method's performance do point toward a broader concern regarding deep learning–namely, its high sensitivity to the availability of training data. Due to limitations in computational resources, we were constrained in the number of graphs we were able to use to train our solver. A greater amount of computing power would enable use of a greater number of training samples, followed by evaluation on a greater number of test samples. We were also constrained in the ability to expand the capacity of our replay-memory, which would further improve learning by enabling more carefully-chosen actions based upon a far greater number of previous actions. Other potential modifications that could further improve results could include a more finely-tuned selection of parameters, including learning-rate. Future work will therefore entail parameter selection to produce better results.

An envisioned use-case of our method would arise in the case of datasets comprising strings with sequences of repeated characters and/or strings of variable length. In these cases, any given action may result in a partial solution that takes, as proper substrings, the entirety of one-or-more strings that have not yet been inserted into the growing superstring. In the case of variable-length strings, this is especially problematic for a greedy approach, since, while the insertion of these proper substrings results in no net increase in cost over the long-term, the level of overlap of these proper substrings may not be particularly large. As an obvious example, consider the partial solution 'AAGAATAATAAGTGA' and the choice between two substrings 'TGA' and 'GTGAG'. A myopic, greedy approach that considers only the length of the current solution's overlap with each substring would select the latter, which may preclude the subsequent "free addition" of the substring 'TGA.' A better option is therefore to reprocess the graph at each update step by removing vertices corresponding to proper substrings of the current solution. Graphically, this could be envisioned as three-vertex feedforward loops of the forms $a \to b$ and $a \to b \to c$ such that insertion of c to a allows us to insert b at no cost. Our method should obviates the need for this reprocessing. Since it considers the effect of taking actions up to n-steps in the future, it will choose actions that first select these proper substrings despite not necessarily maximizing the immediate reward of the action, thereby effectively employing a less-myopic form of a "cheapest-insertion" heuristic.

Thus, future work will immediately entail the deployment of our method on datasets comprising variable-length data. Since the utility of our framework arises as much from the generation of close-approximated solutions to SCSP instances as it does from an ability to discover new heuristics for problem-solving, the evaluation of our proposed framework on new and less well-studied problems and problem-instances presents an exciting opportunity. In these cases, it will be particularly interesting to interpret the greedy heuristics learned by our framework. Such analysis could potentially enable the identification of novel algorithms that have not previously been characterized.

References

[1] Frieze, A., Szpankowski, W. (1998). Greedy Algorithms for the Shortest Common Superstring That Are Asymptotically Optimal. In *Algorithmica 21(1):21-36*

[2] Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M. (1994). Linear Approximation of Shortest Superstrings. In *JACM 41(4):630-647*

[3] Paluch, K. (2014). Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring. In *arXiv preprint. arXiv:1401.3670*

[4] Tarhio, J., Ukkonen, E. (1988). A greedy approximation algorithm for constructing shortest superstrings. In *Theoretical Computer Science 57(1):131-145*

[5] Mucha, M. (2013) Lyndon words and short superstrings. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 958–972. Society for Industrial and Applied Mathematics*

[6] Jager, G., Zhang, W. (2010). An effective algorithm and phase transitions of the directed Hamiltonian cycle problem. In *Journal of Artificial Intelligence Research 39:663-687*

[7] González-Gurrola L.C., Brizuela C.A., Gutiérrez E. (2004) A Genetic Algorithm for the Shortest Common Superstring Problem. In *Lemaître C., Reyes C.A., González J.A. (eds) Advances in Artificial Intelligence – IBERAMIA 2004. IBERAMIA 2004. Lecture Notes in Computer Science, vol 3315. Springer, Berlin, Heidelberg*

[8] Ukkonen, E. (1990). A linear-time algorithm for finding approximate shortest common superstrings. In *Algorithmica 5(1-4):313-323*

[9] Alanko J., Norri T. (2017) Greedy Shortest Common Superstring Approximation in Compact Space. In *Fici G., Sciortino M., Venturini R. (eds) String Processing and Information Retrieval. SPIRE 2017. Lecture Notes in Computer Science, vol 10508. Springer, Cham*

[10] Ohlebusch, E., Fischer, J., Gog, S. (2010). Cst++. In *Chavez E., Lonardi S. (eds) String Processing and Information Retrieval. SPIRE 2010. Lecture Notes in Computer Science, vol 6393. Springer, Berlin, Heidelberg*

[11] Dai, H., Khalil, E.B., Zhang, Y., Dilkina, B., Song, L. (2017). Learning Combinatorial Optimization Algorithms over Graphs. In *Proceedings of the 31st Conference on Neural Information Processing (NIPS 2017).*

[12] Prates, M., Avelar, P., Lemos, H., Lamb, L.C., Vardi, M.Y. (2018). Learning to Solve NP-Complete Problems: A Graph Neural Network for Decision TSP. In *arXiv preprint. arXiv:1809.02721*

[13] Caldwell, J.R., Watson, R.A., Thies, C., Knowles, J.D. (2018). Deep Optimization: Solving Combinatorial Optimization Problems using Deep Neural Networks. In *arXiv preprint. arXiv:1811.00784*

[14] Crochemore M. et al. (2010) Algorithms for Three Versions of the Shortest Common Superstring Problem. In *Amir A., Parida L. (eds) Combinatorial Pattern Matching. CPM 2010. Lecture Notes in Computer Science, vol 6129. Springer, Berlin, Heidelberg*

[15] Kapusta, A., A. Suh and C. Feschotte (2017). Dynamics of genome size evolution in birds and mammals. In *Proc Natl Acad Sci U S A 114(8): E1460-E1469.*

[16] Escalona, M., Rocha, S., and Posada, D. (2016). A comparison of tools for the simulation of genomic next-generation sequencing data. In *Nature reviews. Genetics, 17(8), 459–469. doi:10.1038/nrg.2016.57*

[17] Ning, K. and Leong, H. (2006). Towards a better solution to the shortest common supersequence problem: The deposition and reduction algorithm. In *BMC Bioinformatics, 7(Supplement 4), S12.*

[18] Mousavi, S.R., Tabataba, F.(2012) An improved algorithm for the longest common subsequence problem In *Computers and Operations Research, 39 (3), pp. 512-520.*

[19] Saifullah KCM, Islam MR (2016) Chemical reaction optimization for solving shortest common supersequence problem. In *Comput Biol Chem 64:82–93*

[20] Grover, A., and Leskovec, J. (2016, August). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 855-864). ACM.*

[21] Chandrasekaran, V., Parrilo, P., and Willsky, A. (2010, Dec). Convex Graph Invariants. In *SIAM Review, Vol. 54, No. 3 (pp. 513-541).0*

[22] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. In *arXiv preprint arXiv:1312.5602.*

[23] Thrun, S., Schwartz, A. (1993, Dec). Issues in Using Function Approximation for Reinforcement Learning. In *Proceedings of the Fourth Connectionist Models Summer School Lawrence Erlbaum Publisher, Hillsdale, NJ*

[24] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In *In Thirtieth AAAI Conference on Artificial Intelligence.*