ORIGINAL ARTICLE

Resonance algorithm: an intuitive algorithm to find all shortest paths between two nodes

Yu Liu[1] · Qiguang Lin[2] · Binbin Hong[3] · Yunru Peng[1] · Daniel Hjerpe[4] · Xiaofeng Liu[2,5]

Abstract

The shortest path problem (SPP) is a classic problem and appears in a wide range of applications. Although a variety of algorithms already exist, new advances are still being made, mainly tuned for particular scenarios to have better performances. As a result, they become more and more technically complex and sophisticated. In this paper, we developed an intuitive and nature-inspired algorithm to compute all possible shortest paths between two nodes in a graph: *Resonance Algorithm* (RA). It can handle any undirected, directed, or mixed graphs, irrespective of loops, unweighted or positively weighted edges, and can be implemented in a fully decentralized manner. Although the original motivation for RA is not the speed per se, in certain scenarios (when sophisticated matrix operations can be employed, and when the map is very large and all possible shortest paths are demanded), it out-competes Dijkstra's algorithm, which suggests that in those scenarios, RA could also be practically useful.

Keywords Dijkstra's algorithm · Large-scale graph · Nature-inspired algorithm · Decentralized algorithm · Fermat's principle

Introduction

The shortest path problem (SPP), i.e., to find a path between two nodes in a graph such that the length (or weights) of the path is minimized, is a classic problem in graph theory and computer science [1], with a wide range of applications such as route planning [2–4], network routing [5–7] and task planning [8,9]. There is a variety of algorithms to solve SPP, including the classic Dijkstra's algorithm [10] (which

Yu Liu and Qiguang Lin have contributed equally to this work.

✉ Yu Liu
  yu.ernest.liu@bnu.edu.cn

✉ Xiaofeng Liu
  xfliu@hhu.edu.cn

1 International Academic Center of Complex Systems, Beijing Normal University, Zhuhai 519087, China

2 College of IoT Engineering, Hohai University, Changzhou 213022, China

3 Institute of Microscale Optoelectronics, Shenzhen University, Shenzhen 518060, China

4 Ericsson AB, Kista 16483, Sweden

5 Jiangsu Key laboratory of Special Robotic Technologies, Changzhou 213022, China

is commonly used and lays the basis for many other methods [11–13]), Bellman–Ford algorithm [14], Chan's fast algorithm (in $O(n^3/\log n)$ time) [15], etc.

Although SPP has been studied extensively, new advances are still being made to have better performances in specific scenarios, including rapid explorations [16,17], solving multi-objective problems [18,19], handling dynamic networks [20–22] and stochastic situations [23,24]. More specifically, for example, Noto and Sato proposed a fast algorithm to compute paths as close as possible to the optimal solution (much faster than the algorithms that give optimal solutions) to handle real-time problems [17]. Galán-García et al. developed Probabilistic Extension of Dijkstra's Algorithm to calculate not only the shortest path but also the second-, third- or fourth-shortest path, by taking the traffic flows into account, which is extremely helpful in realistic car navigation [25]. Moreover, by taking off the "Fibonacci heap" in the original Dijkstra's algorithm, Xu et al. substantially improved the efficiency of Dijkstra's algorithm for sparse networks, to which the road traffic network belongs [26].

While finding one shortest path has a wide range of applications (as in the above examples), finding all possible shortest paths (without missing any one) is also a significant question to ask in various scenarios [27], e.g., when searching a path that should satisfy other conditions beyond having

the minimal length such as in the power line routing problem [28]. For another example, in the biological sequence alignment problem (which can be reduced to SPP), we need to investigate alternative optimal solutions in order to check the sensitivity of this problem [29]. Moreover, researchers developed a computational approach to identify liver cancer related genes, where the essential step is to find all (not just only one) shortest paths in the protein-protein interaction network [30].

The most straightforward approach to find all shortest paths is to simply use breadth-first search, which is inefficient though [31]. Alternatively, Dijkstra's algorithm can be extended to return all possible shortest paths [32]. We can also "hack" the algorithm that solves the *k shortest paths problem* [33] (i.e., to list *k* paths in ascending lengths where *k* is a predefined positive integer parameter) so that *k* is not predefined and the algorithm continues to iterate until the newly returned path is strictly longer than the path returned in the last iteration.

In this paper, we propose an intuitive and intriguing algorithm that finds all possible shortest paths between two nodes in a graph. We call this algorithm *Resonance Algorithm* (RA). First, we recommend the reader to check out the 50-s animation of RA at www.wuyichen.org/resonance-algorithm, for a general idea. Generally speaking, RA can be summarized in three points:

1. Starting from the origin, each node sends signals to all of its neighbors when it receives one for the first time, until the destination receives the signal, which we call the *forward process*;
2. Likewise, we have the *backward process* but the direction is from the destination to the origin;
3. All possible shortest paths are the intersections of the forward process and the time-reversed backward process.

Note that the "signal" is just something imagined that travels in the graph, which could be anything, e.g., an empty message. It serves as a metaphor that may help the readers better understand the logic of the algorithm. Intriguingly, RA might be considered as a metaphor of the combination of Fermat's principle [34] (i.e., the path taken by a ray is always the one that takes the least time) and the probability amplitude interpretation of the wave function in quantum mechanics [35] (see discussions in Supplementary Notes online).

There are a variety of nature-inspired optimization algorithms [36], some of which could be adopted to handle SPP, such as the ant colony algorithm [37,38], the simulated annealing algorithm [39], the fruit fly optimization algorithm [40], the whale optimization algorithm [41] and the bald eagle search algorithm [42]. RA that we proposed here would belong to those nature-inspired algorithms, but designed particularly for SPP. The motivation of RA is not to provide a

time-efficient algorithm *per se* (although we later find that it is fast in particular scenarios), but to serve as an alternative nature-inspired algorithm, providing a new look at the classic question.

This paper is organized as follows. In the next section, we go through a typical example to illustrate the general scheme of RA. In the third section, we explain in details about how to implement RA by matrix, one of many possible implementations. Next, in the fourth section, we compare RA with Dijkstra's algorithm mechanistically in details, and then statistically in time efficiency. In the end, we discuss the potential applications of RA and briefly discuss how to implement RA in a decentralized manner.

Resonance algorithm (RA): general scheme

To illustrate RA, take the undirected and weighted graph shown in Fig. 1a as an example (the weighting is represented by the length of edges). Now, the following three subsections elaborate RA step by step, also referring to the animation mentioned above www.wuyichen.org/resonance-algorithm. The example in Fig. 1 is the simplest case, but refer to the Supplementary Notes online for the example with mixed edges (namely with both undirected and direct edges) and weighted edges, irrespective of loops, which represents the most general case.

Forward process: from origin to destination

- To begin with ($t = 0$), the origin node A sends signals to all of its neighbors simultaneously, namely nodes C and S.
- At $t = 1$, nodes C and S receive the signal. Immediately, C sends signals to its neighbors D and E; and S sends signals to its neighbors R and Q. Note that they do not need to send signals to the node where the signal came from.
- At $t = 2$, D, E, R and Q receive the signal. Immediately, D sends a signal to E; E sends signals to D and F; R sends a signal to H; and Q sends signals to N and P.
- At $t = 3$, E, D, F, H, N and P receive the signal. Note that, although D and E receive signals again, they do not need to send signals because this is not the first time they receive them (which means D and E will never send signals again). Nodes F, H, N, and P send signals to their neighbors, immediately.
- This process continues, until the destination node B receives a signal. This is then the *forward process*, denoted as $\mathcal{X}$.
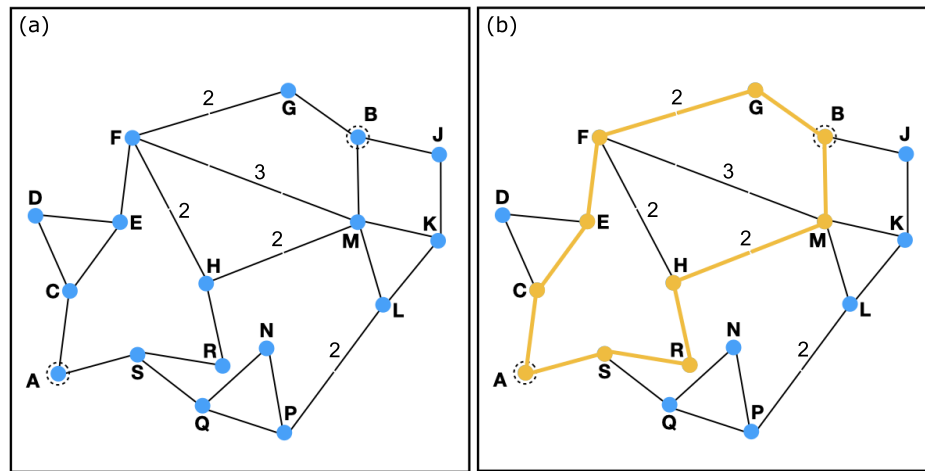
Fig. 1 The exemplified graph to illustrate RA, also referring to the animation at www.wuyichen.org/resonance-algorithm. **a** The question is to find all possible shortest paths from the origin node A to the destination node B. This graph is undirected and weighted, represented by lengths of the edges (weights that are larger than 1 has been explicitly written down, otherwise omitted), which can be interpreted as the time needed to move from one end to the other, e.g., it takes 1 unit of time (e.g., second) to move from A to C, 1 second from R to H, 2 s from M to H and 3 s from F to M. Here, the weights are assumed to be integers but they could be any positive values. **b** The highlighted yellow paths are the two and the only two shortest paths from node A to B

Backward process: from destination to origin

In the *backward process*, the signal is sent from the destination node B to the origin node A, but all of the nodes obey the exact rules as in the forward process. We denote the backward process as $\mathcal{Y}$. In fact, it does not matter if the forward or the backward process is implemented first. If parallel computing is allowed, we could simultaneously implement both.

Intersection of forward and backward processes

This is the last step of RA, after which all possible shortest paths will reveal themselves:

- In this step, we simultaneously play the animation of the forward process $\mathcal{X}$ and the backward process $\mathcal{Y}$. But note that the key is to play $\mathcal{X}$ normally but play $\mathcal{Y}$ reversely, namely in a time-reversed manner.
- At each frame during playing, we mark the signals that appear at the same position in both processes.
- In the end, the paths covered by all of these marked signals are the shortest paths (referring to the highlighted yellow paths in the animation, also shown in Fig. 1b).

Resonance algorithm (RA): implemented by matrix

In this section, we explain in details about how to implement RA by matrix, one of many possible implementations,

also referring to the MATLAB code at https://github.com/yuernestliu/ResonanceAlgorithm.

Forward process: from origin to destination (matrix)

First, we initialize a zero matrix (denoted as **X** and each entry is denoted as $X_{i,j}$) where one row stands for one node and one column stands for one time point. **X** records the time point when a node sends signals, where "1" represents signals are sent and "0" otherwise.

At $t = 0$, we set $X_{A,0} = 1$, meaning that the origin node A sends signals to all of its neighbors. Its neighbors C and S will receive this signal at $t = 1$ (as the weights of both edges are 1). We then set $X_{C,1} = X_{S,1} = 1$, as shown in Fig. 2a ("0" is always omitted to write).

Now, repeat the following two procedures until the destination node B receives signals: (i) set the time to $t + 1$, and check the whole column of $(t + 1)$ to see which entries are 1, meaning that they receive signals at $(t + 1)$ and will send signals to all of their neighbors immediately (denote each neighbor as v); (ii) for each v, there is a specific time point (denoted t_v) it receives the signal, so we mark each $X_{v,t_v} = 1$. But note that (1) if $X_{v,\tau} = 1$ for any $\tau < t_v$ (meaning that v would send signals before t_v), do not mark it; and (2) if v will receive signals at several time points after $(t + 1)$, only mark the earliest time point and change other entries to 0. Specifically:

- At $t = 1$, we have $X_{C,1} = X_{S,1} = 1$ which means that C and S receive signals at $t = 1$ (Fig. 2b) and they will
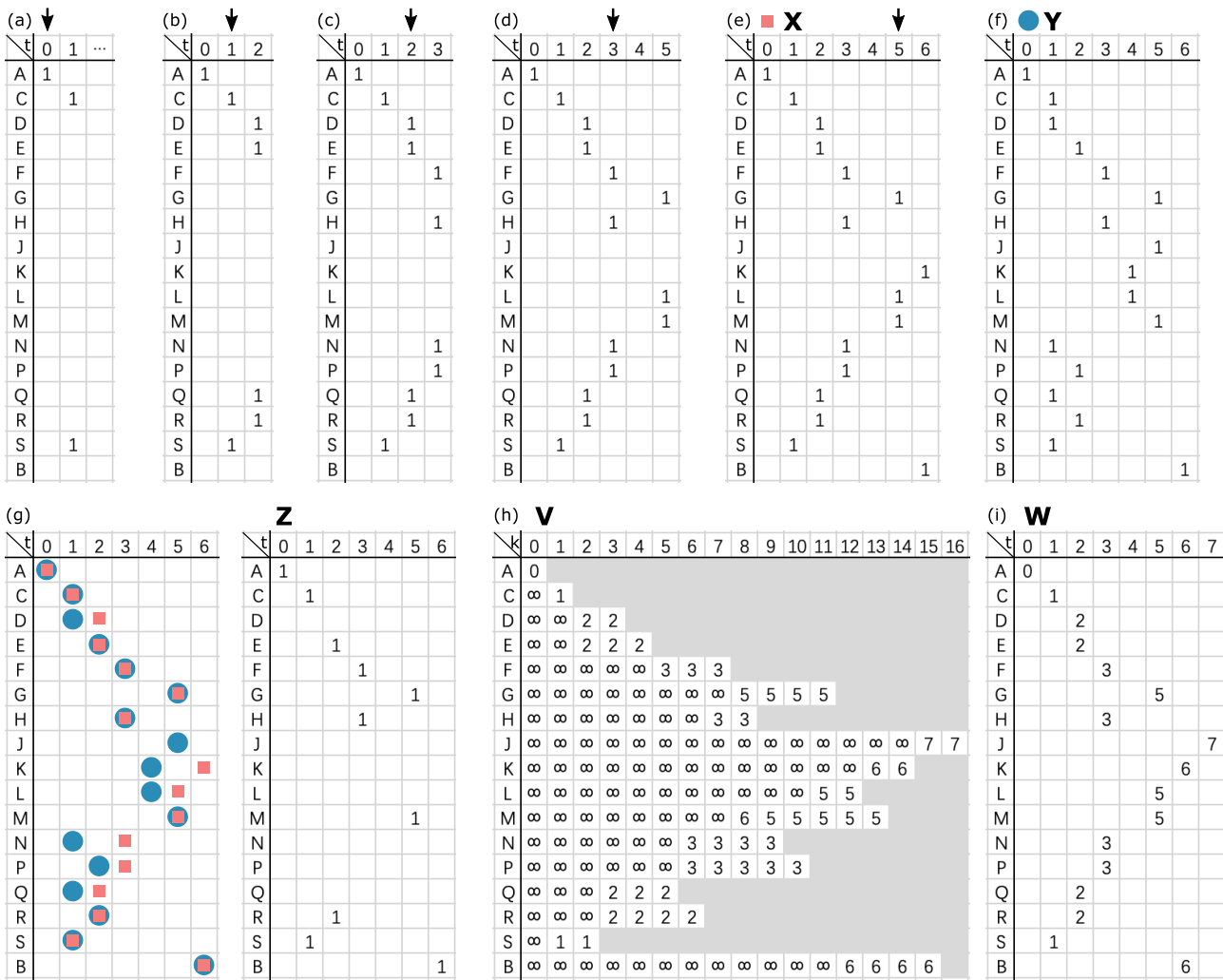
Fig. 2 The matrix implementation of RA ("0" is omitted in these matrices). **a–e** Illustrate how to obtain the matrix **X** for the forward process. **f** Shows the matrix **Y** for the backward process. **g** Illustrates the intersection operation where the red squares represent the entries where $X_{i,j} = 1$, and the blue circles represent the entries where $Y_{i,j} = 1$. The entries with both a red square and a blue circle will be set to 1.

Then we obtain the matrix **Z**. **h** The matrix implementation of Dijkstra's algorithm for the same graph (Fig. 1a). The entry records the tentative distance from the origin to this node, and the ones marked in gray indicate that they are visited. **i** To compare with RA, we rewrite the matrix **V** in terms of time points t instead of operation steps k, into the matrix **W**

send signals to their neighbors immediately. For node C, its neighbors D and E will receive signals at $t = 2$, as the weights of both edges are 1, so we mark $X_{D,2} = X_{E,2} = 1$ (although node A is also C's neighbor, it has already sent signals, so leave it). Similarly, for node S, we mark its neighbors $X_{Q,2} = X_{R,2} = 1$.

- At $t = 2$, we see that D, E, R and Q receive the signals (Fig. 2c), and will send signals to their neighbors immediately. Therefore, we mark their corresponding neighbors at the time point they receive signals, namely $X_{F,3} = X_{H,3} = X_{N,3} = X_{P,3} = 1$.
- At $t = 3$, F, H, N and P receive the signals (Fig. 2d). Likewise, we mark their corresponding neighbors at the corresponding time point $X_{G,5} = X_{L,5} = X_{M,5} = 1$

(notice the weighted edges, i.e., it takes 2 units of time from F to H and G, from H to F and M, from P to L, and it takes 3 units of time from F to M). Note that the signal that F sent arrives at M at $t = 6$ while the signal that H sent arrives at M at $t = 5$, but we will only mark the one that arrives first, i.e., only set $X_{M,5} = 1$.

- At $t = 4$, no node receives signals, so we directly move to the next time point.
- At $t = 5$, G, L and M receive signals for the first time (Fig. 2e). Likewise, we mark their corresponding neighbors $X_{B,6} = X_{K,6} = 1$.
- At $t = 6$, the destination node B receives the signal, so the forward process stops. We then obtained the matrix **X** for the forward process, as shown in Fig. 2e.

Backward process: from destination to origin (matrix)

In the backward process, the rules each node follows are exactly the same as that in the forward process, except that (1) the signal is sent from the destination B to the origin A, and that (2) the time goes backwards, i.e., t starts from 6 (the time point when node B receives the signal in the forward process), to 5, 4, ..., till 0. It is automatically guaranteed that at $t = 0$, the origin A will receive the signal. In the end, we can obtain the matrix $\mathbf{Y}$ for the backward process, as shown in Fig. 2f.

Intersection of forward and backward processes (matrix)

The last step is to compute the intersection of $\mathbf{X}$ and $\mathbf{Y}$: Create a matrix $\mathbf{Z}$ and set each entry of $\mathbf{Z}$ to be equal to the logical conjunction (AND) of the corresponding entries of $\mathbf{X}$ and $\mathbf{Y}$. That is, $Z_{i,j} = X_{i,j}$ AND $Y_{i,j}$, i.e., 1 AND 1 = 1, while 0 otherwise. The matrix $\mathbf{Z}$ is all we need, that contains the information of all possible shortest paths (Fig. 2g). From $\mathbf{Z}$, we simply trace back how the signal is propagated and we can obtain all the shortest paths. Since the intersection only contains the information of the shortest paths (if there is only one shortest path, the intersection only contains the information of this particular path; if there are two shortest paths, the intersection only contains the information of these two particular paths; and so forth) and thus has filtered out all the redundant information, the process of tracing back would be super fast. The paths can be visualized as shown in Fig. 1b.

Comparison with Dijkstra's algorithm

Dijkstra's algorithm is the most classic and commonly used algorithm for SPP [10,11,43]. One difference between Dijkstra's algorithm and RA is that the latter returns all possible shortest paths while the former (the classic version) returns one shortest path (yet extensions can be made to return all possible shortest paths [32]).

Therefore, in this section, we will first take the classic version of Dijkstra's algorithm, and compare it with RA for similarities and differences in the searching process. And then, we will extend the classic Dijkstra's algorithm to return all possible shortest paths, and compare the running time of the classic and extended Dijkstra with RA.

Comparison with Dijkstra's algorithm: searching process

In general, RA and Dijkstra's algorithm have a few points in common: (1) If a node is currently in focus, the next step is to check all of its neighbors: the former does it by sending signals to all of its neighbors, while the latter does it by updating the distance from the origin to the neighbor. (2) They both ignore certain nodes in future searching processes: RA ignores the nodes that have sent signals, while Dijkstra's algorithm ignores the nodes that have been visited. (3) RA stops searching as long as the destination receives signals, while Dijkstra's algorithm stops when all nodes have been visited (but note that if we only need the shortest path between the origin and the destination, instead of between the origin and all other nodes as in the classic Dijkstra, early exit can also be applied to Dijkstra, to speed up).

To compare the two algorithms, we now apply Dijkstra's algorithm on the exemplified graph in Fig. 1a, and use the similar matrix notation as in "Resonance algorithm(RA): implemented by matrix". First, we initialize a matrix $\mathbf{V}$ where one row stands for one node, and one column stands for one iteration, denoted k (referring to the matrix shown in Fig. 2h, which evolves as the algorithm proceeds, but we currently only look at column $k = 0$). Each entry of the matrix records the *tentative distance* from the origin to this node. For column $k = 0$ (initialization), we set the entry for the origin A to 0, and other entries to infinity ∞. Mark all nodes as *unvisited*, whereas *visited* nodes are marked in grey as we shall see later.

- We are first at the initial operation step (column $k = 0$). Select one unvisited node that is marked with the smallest tentative distance, node A in this case. Consider all of its unvisited neighbors, nodes C and S in this case, and calculate their tentative distances, both of which are $0 + 1 = 1$ where 0 is the current tentative distance for A, and 1 is the weight of the edge from A to C or S. Since the new tentative distance (namely 1) is smaller than the current value ∞, we update the corresponding entries of the next operation step to 1, i.e., $V_{C,1} = V_{S,1} = 1$. Then, at column $k = 1$, we mark node A visited (denoted by grey color, and A will not be visited again) and keep other entries unchanged. Finally, matrix $\mathbf{V}$'s column $k = 1$ is updated.
- Now, we are at column $k = 1$. Select one unvisited node with the smallest tentative distance. Either C or S works, and we can choose C first. Consider all of C's unvisited neighbors (namely D and E), and calculate their tentative distances, both being $1 + 1 = 2$. Since 2 is smaller than the current value ∞, update the entries $V_{D,2} = V_{E,2} = 2$ (and other entries unchanged). Finally, mark node C visited.

- Then, we are at column $k = 2$. Select one unvisited node with the smallest tentative distance, S in this case. Calculate the tentative distances of all of S's unvisited neighbors (namely Q and R), both being $1 + 1 = 2 < \infty$. Then update $V_{Q,3} = V_{R,3} = 2$, and mark S visited.
- This process continues, until all node are marked visited. As we see in Fig. 2h, it finishes at $k = 16$ in this case.

In order to compare **V** with the matrix obtained from RA, we rewrite **V** in terms of time points t instead of operation steps k, into the matrix **W** (Fig. 2i), that is, for each node, if the rightmost entry is m in **V**, we write m in **W** at the mth column and the corresponding row.

We can see that **W**'s non-empty entries and **X**'s value-1 entries have exactly the same positions (**W** has an extra 7th column due to the fact that all nodes have to be visited in Dijkstra's algorithm). That means, the forward process of RA (as **X** is the resulted matrix) and Dijkstra's algorithm are similar, essentially. Two algorithms would thus have a similar algorithmic complexity. One minor difference between the two algorithms is that RA will stop right after the destination receives the signal, while Dijkstra's algorithm will go through every nodes in the map (although early exit can be applied if we only need one shortest path between the origin and the destination).

The major difference comes after we obtain this above-mentioned matrix. Dijkstra's algorithm finds the shortest path by following the parent nodes up to the origin as every searched node has been marked with a parent node, while RA computes the intersection of the forward and the backward process.

It may sound redundant to run a backward process (although there is no effect on the time complexity as it only doubles the computing time), but exactly because of this, RA does not require to either record and update each node's tentative distance or memorize the paths visited (which may reduce the computing time). Yet, it is exactly this symmetrical operation of the forward and time-reversed backward process that makes RA intriguing and intuitive. And it is a natural consequence that after the operation of the intersection, all shortest paths are revealed.

Comparison with Dijkstra's algorithm: running time

In order to have a quantitative comparison, we will systematically investigate their running times here. Note that the classic Dijkstra's algorithm only gives one shortest path between two nodes. Therefore, we extend it so that it gives all possible shortest paths between two nodes [32] (see Supplementary Notes online for how to extend Dijkstra's algorithm), and then we compare the classic and the extended Dijkstra's algorithm with RA. For all of these algorithms, we wrote the codes in MATLAB with sufficient optimization, e.g.,

we allow early exit for Dijkstra (see the codes on https://github.com/yuernestliu/ResonanceAlgorithm). Now we run the codes on different random graphs with the number of nodes varying (see Supplementary Notes online for how these random graphs are generated).

First, we have confirmed that RA and the extended Dijkstra's algorithm always give identical answers, and that the shortest paths given by the classic Dijkstra's algorithm are always included in the shortest paths given by RA. That is, we confirm that RA always give correct answers.

Then, we show the running time statistics. We first run the code on Octave (an open-sourced software compatible with MATLAB). We can see from Fig. 3a that RA is slower than Dijkstra (both the classic and the extended), which is reasonable because (1) as we have discussed above, essentially RA and Dijkstra have similar complexity, as long as we optimize Dijkstra so that it can have early exit (without early exit, Dijkstra runs much slower than RA when the graph is large), and (2) RA requires a backward process.

But what is surprising is that when we run the same code on MATLAB, we can see from the results, as shown in Fig. 3b, that RA out-competes Dijkstra (both the classic and the extended) when the graph is relatively large (around 50 nodes in this case) and its running time grows much slower than Dijkstra. This might be due to the sophisticated optimization of matrix operations in MATLAB, as RA involves much more matrix operations than Dijkstra, which can be seen in "Resonance algorithm(RA): implemented by matrix". Therefore, although the original motivation for RA is only to give a new and intuitive algorithm that is inspired by natural processes, instead of the speed *per se*, it could be practically useful in some scenarios, e.g., when sophisticated matrix operations can be employed, and when the map is very large and all possible shortest paths are demanded.

Discussion

In this paper, we developed an intuitive algorithm, *Resonance Algorithm* (RA), to compute all of the possible shortest paths between two nodes in a graph, which is inspired by Fermat's principle (i.e., the path taken by a ray is always the one that takes the least time) and the probability amplitude interpretation of the wave function in quantum mechanics, as the wave function always experiences every possible path (see Supplementary Notes online for a discussion).

The basic idea of RA is that if each node sends signals to all of its neighbors immediately after it receives one, then when the destination receives the signal for the first time, the signal must have traveled through the shortest path to reach the destination. The information about the shortest paths has already been stored in this process, yet the problem is how to extract it (the well-known Dijkstra's algorithm does it by updating

Fig. 3 Running time statistics of the classic, the extended Dijkstra's algorithm and RA. The curve is the mean running time while the error bar represents the standard error. For a fixed number of nodes (*x*-axis value), we used 50 randomly generated graphs, all of which are sparse (and connected) graphs, i.e., the number of edges is much smaller than the number of nodes squared, of which the graph in Fig. 1a is a typical example. **a** The code was run on Octave on a personal computer with the Intel(R) Core(TM) i7-7700 CPU of 3.60 GHz, with 16 GB RAM. **b** The same code was run on MATLAB on the same computer
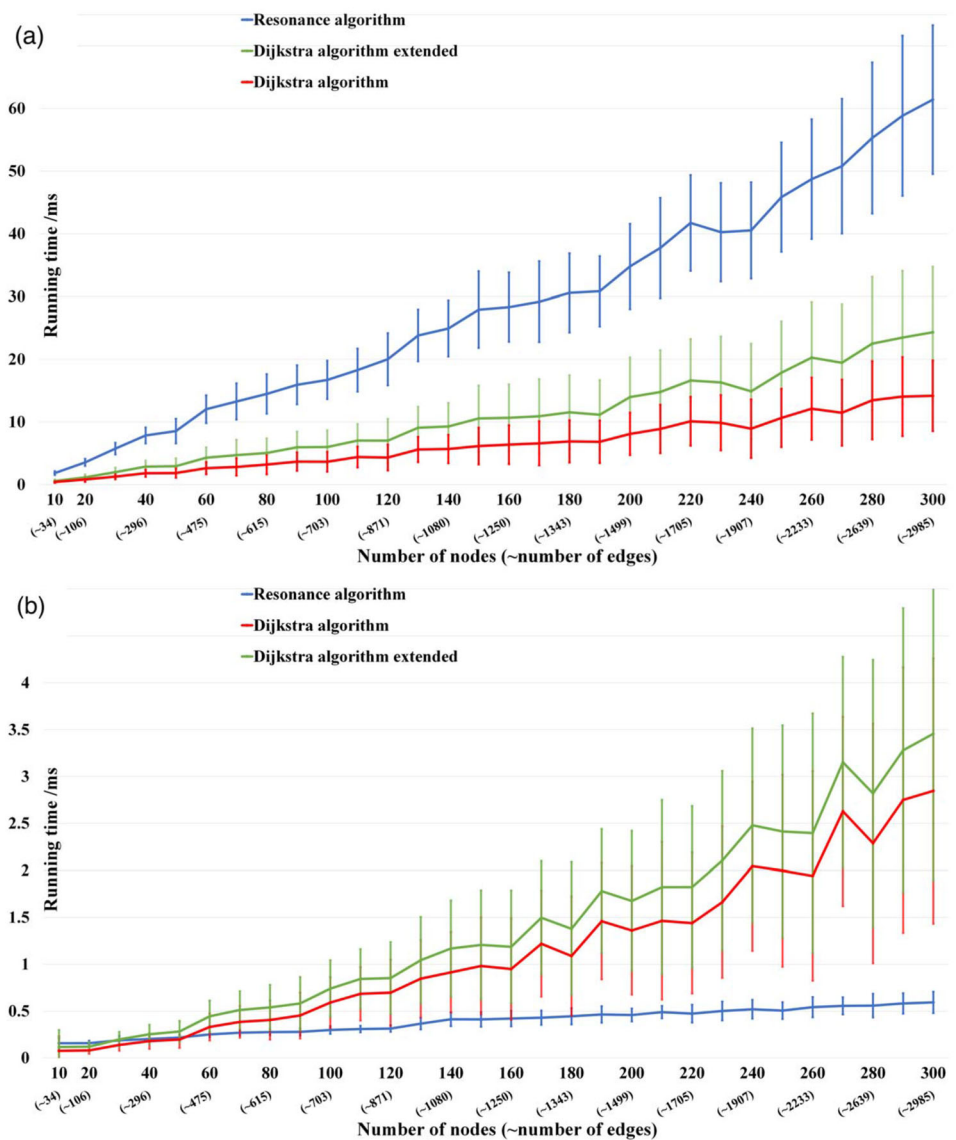
the tentative distance from the origin to the neighbor, and taking notes of the path it travels). On the other hand, if the signal is sent from the destination to the origin, there must be signals traveling through the shortest path, too. Therefore, the intersection of the forward process and the time-reversed backward process may reveal the shortest paths. This is the most basic motivation.

Practically speaking, RA can handle any undirected, directed, or mixed graphs, with or without loops, with unweighted or positively weighted edges. In addition, RA does not require the information of the whole graph to be stored in one central agent, and it can thus be implemented in a fully decentralized manner, because each node acts as an independent agent, obeys identical rules, and its behavior depends only on the local information, i.e., which node it is linked to and when it receives signals. That is, the nodes collectively determine the shortest paths. This would be very

useful in scenarios where the central agent does not exist or cannot keep track of the whole graph, or where nodes are constantly joining and leaving the graph.

Here, we provide a recipe for the decentralized version. First, each node only records who it is linked to, namely all of its neighbors (so there is no need for the whole graph to be hold by a central agent). Second, unlike in the normal version where the "signal" could be an empty message, for this decentralized version, the "signal" must contain certain information: That is, when a signal is about to be sent, the information about when it is sent and which neighbor it is sent to should be added into the signal itself. Then, when the signal finally arrives at the destination (denoted as signal A), the propagation history of this signal would be fully recorded in itself (it is straightforward to see that each signal only records the propagation history of itself and thus differs from each other). This also applies to the backward process,

that is, when the signal arrives at the origin (from the destination), its propagation history would also be fully recorded in itself (denoted as signal B). Then, signal B should be sent back to the destination node for computing (or maybe other computing units). When both signals A and B are obtained by the destination node (the computing unit in this example), the matrix $\mathbf{X}$ and $\mathbf{Y}$ (see Fig. 2) can then be reconstructed, from which the shortest paths can be readily worked out (referring to "Resonance algorithm(RA): implemented by matrix" where we described how to do it in details).

The final remark is that the original motivation for RA is just to give an intuitive and nature-inspired algorithm, providing a new look at SPP, but in certain scenarios it out-competes the classic and the extended version of Dijkstra's algorithm in time efficiency. It suggests that RA could also be practically useful when sophisticated matrix operations can be employed, and when the map is very large and all possible shortest paths are demanded.

Data availability The codes and datasets generated and analyzed during the current study are available in the GitHub repository https://github.com/yuernestliu/ResonanceAlgorithm.

Declarations

Conflict of interest The authors declare that they have no competing interests.

References

1. Deo N, Pang C-Y (1984) Shortest-path algorithms: taxonomy and annotation. Networks 14:275–323
2. Zhang X, Mahadevan S (2018) A bio-inspired approach to traffic network equilibrium assignment problem. IEEE Trans Cybern 48(4):1304–1315
3. Bast H, Delling D, Goldberg A, Müller-Hannemann M, Pajor T, Sanders P, Wagner D, Werneck RF (2016) Algorithm Engineering. Route Planning in Transportation Networks, pp 19–80. Springer, Cham
4. Delling D, Sanders P, Schultes D, Wagner D (2009) Algorithmics of Large and Complex Networks. Engineering Route Planning Algorithms, pp 117–139. Springer, Berlin
5. Medhi D, Ramasamy K (2018) Chapter 2 - Routing Algorithms: Shortest Path, Widest Path, and Spanning Tree, Second edition edn. The Morgan Kaufmann Series in Networking, pp 30–63. Morgan Kaufmann, Boston
6. Mahboubi H, Masoudimansour W, Aghdam AG, Sayrafian-Pour K (2017) An energy-efficient target-tracking strategy for mobile sensor networks. IEEE Trans Cybern 47(2):511–523
7. Dijkstra F, van der Ham J, Grosso P, de Laat C (2009) A path finding implementation for multi-layer networks. Fut Gen Comput Syst 25:142–146
8. Sun X, Liu Y, Yao W, Qi N (2015) Triple-stage path prediction algorithm for real-time mission planning of multi-UAV. Electron Lett 51:1490–1492
9. Xiao P, Ju H, Li Q, Xu H, Lu C (2020) Task planning of space maintenance robot using modified clustering method. IEEE Access 8:45618–45626
10. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1:269–271
11. Peng W, Hu X, Zhao F, Su J (2012) A fast algorithm to find all-pairs shortest paths in complex networks. Procedia Comput Sci 9:557–566
12. Lu X, Camitz M (2011) Finding the shortest paths by node combination. Appl Math Comput 217(13):6401–6408
13. Orlin JB, Madduri K, Subramani K, Williamson M (2010) A faster algorithm for the single source shortest path problem with few distinct positive lengths. J Discrete Algorithm 8(2):189–198
14. Bang-Jensen J, Gutin G (2009) Digraphs: Theory, Algorithms and Applications, pp. 55–58. Springer, London. Chap. The Bellman-Ford-Moore algorithm
15. Chan TM (2008) All-pairs shortest paths with real weights in $o(n^3 / \log n)$ time. Algorithmica 50(2):236–243
16. Liu X, Li J, Hu F, Yang C (2019) Obstacle induced stochastic tree for fast path planning. In: 2019 IEEE International Conference on Real-time Computing and Robotics (RCAR), pp 499–503
17. Noto M, Sato H (2000) A method for the shortest path search by extended Dijkstra algorithm. In: 2000 IEEE international conference on systems, man and cybernetics, pp 2316–2320
18. Sastry VN, Janakiraman TN, Mohideen SI (2003) New algorithms for multi objective shortest path problem. Opsearch 40:278–298
19. Storandt S (2012) Quick and energy-efficient routes: Computing constrained shortest paths for electric vehicles. In: Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science, pp. 20–25. Association for Computing Machinery, New York, NY, USA
20. Zhang S, Liu X (2013) A new algorithm for finding the k shortest transport paths in dynamic stochastic networks. J Vibroeng 15:726–735
21. Zhu D, Huang H, Yang SX (2013) Dynamic task assignment and path planning of multi-AUV system based on an improved self-organizing map and velocity synthesis method in three-dimensional underwater workspace. IEEE Trans Cybern 43(2):504–514

22. Moon S, Oh E, Shim DH (2013) An integral framework of task assignment and path planning for multiple unmanned aerial vehicles in dynamic environments. J Intell Robot Syst 70(1):303–313

23. Pascoal MMB, Resende M (2014) The minmax regret robust shortest path problem in a finite multi-scenario model. Appl Math Comput 241:88–111

24. Ohtsubo Y (2008) Stochastic shortest path problems with associative accumulative criteria. Appl Math Comput 198(1):198–208

25. Galán-García JL, Aguilera-Venegas G, Galán-García MÁ, Rodríguez-Cielos P (2015) A new probabilistic extension of Dijkstra's algorithm to simulate more realistic traffic flow in a smart city. Appl Math Comput 267:780–789

26. Xu MH, Liu YQ, Huang QL, Zhang YX, Luan GF (2007) An improved Dijkstra's shortest path algorithm for sparse network. Appl Math Comput 185(1):247–254

27. Hershberger J, Maxel M, Suri S (2007) Finding the k shortest simple paths: a new algorithm and its implementation. ACM Trans Algorithms 3:45

28. Hemalatha S, Valsalal P (2012) Identification of optimal path in power system network using bellman ford algorithm. Model Simul Eng 2012:913485

29. Waterman MS (1983) Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. Proc Natl Acad Sci 80:3123–3124

30. Jiang M, Chen Y, Zhang Y, Chen L, Zhang N, Huang T, Cai Y-D, Kong X (2013) Identification of hepatocellular carcinoma related genes with k-th shortest paths in a protein-protein interaction network. Mol BioSyst 9:2720–2728

31. Eppstein D (1998) Finding the k shortest paths. SIAM J Comput 28:652–673

32. Wang S, Li A (2014) Multi-adjacent-vertexes and multi-shortest-paths problem of Dijkstra algorithm. Comput Sci 41:217–224

33. Mohanta K (2012) Comprehensive study on computational methods for k-shortest paths problem. Int J Comput Appl 40:22–26

34. Ziggelaar A (1980) The sine law of refraction derived from the principle of Fermat—prior to Fermat? The theses of Wilhelm Boelmans S. J. in 1634. Centaurus 24, 246–262

35. Schlosshauer M (2005) Decoherence, the measurement problem, and interpretations of quantum mechanics. Rev Mod Phys 76:1267–1305

36. Yang X-S, He X-S (2019) Nature-inspired algorithms, pp 21–40. Springer, Cham

37. Dorigo M, Birattari M, Stutzle T (2006) Ant colony optimization. IEEE Comput Intell Magn 1(4):28–39

38. Dorigo M, Blum C (2005) Ant colony optimization theory: a survey. Theor Comput Sci 344(2):243–278

39. Guilmeau T, Chouzenoux E, Elvira V (2021) Simulated annealing: a review and a new scheme. In: 2021 IEEE Statistical Signal Processing Workshop (SSP), pp 101–105

40. Pan WT (2012) A new fruit fly optimization algorithm: taking the financial distress model as an example. Knowl-Based Syst 26(2):69–74

41. Mirjalili S, Lewis A (2016) The whale optimization algorithm. Adv Eng Softw 95:51–67

42. Alsattar HA, Zaidan AA, Zaidan BB (2020) Novel metaheuristic bald eagle search optimisation algorithm. Artif Intell Rev 53(6):2237–2264

43. Helgason RV, Kennington JL, Stewart BD (1993) The one-to-one shortest-path problem: an empirical analysis with the two-tree Dijkstra algorithm. Comput Optim Appl 2:47–75