International Journal of Advanced Science and Technology
Volume 5, April, 2009

Kohonen-guided Parallel Bidirectional Voronoi-assisted Heuristic Search

Anestis A. Toptsis, Rahul A. Chaturvedi, and Avaz Feroze
Dept. of Computer Science and Engineering,
York University, Toronto, Ontario, Canada
{anestis, rahulac, avaz}@yorku.ca

Abstract

Multi-process bidirectional heuristic search algorithms (such as PBA) that utilize island nodes have been shown to have the potential for exponential speedup over their plain counterparts that do not utilize island nodes. PBA* (Parallel Bidirectional A*) is a primary algorithm in this area and, to the best of our knowledge, unique in its usage of island nodes for reducing the size of the traversed search space and in its usage of multidimensional heuristics for reducing interprocess communication cost. Both, island nodes use and reference nodes use within the context of multidimensional heuristics have been reported to provide excellent performance, compared to plain heuristic search counterparts. However, both of these two features have also resisted refinements for over two decades. Specifically, the two main issues are: how to identify well-placed (in the state space) island nodes, and how to generate well-placed reference nodes. The work presented in this paper is an initial step toward this end. We present two methods, one employing Voronoi diagrams for the purpose of identifying well-placed island nodes, and one employing Kohonen Networks for the purpose of identifying well-placed reference nodes. We implement and compare our methods against a plain version of PBA*. Our finding indicate that the cost of incorporating the proposed improvements into PBA* is negligible; and when PBA* is equipped with the island identification method, or with the reference node identification method, it outperforms its random island nodes counterpart and its random reference nodes counterpart, respectively, for the vast majority of test cases.*

1. Introduction and Background

Heuristic search is one of the foundational areas of artificial intelligence (AI). Recently, it has many applications in several diverse and practical areas such as Software Engineering and the Web (e.g., [1, 2, and 3]). Heuristic search is essentially a graph traversal with the characteristic that the number of nodes in the graph is *huge* and thus the graph nodes are generated on the fly (as opposed to being already known), as we traverse the graph. This sort of traversal is called heuristic search, because we always search for some "goal" node G, in the graph, and we always use heuristics (guesses and estimates that indicate the best way to proceed in the course of the graph traversal). Heuristic search algorithms can be classified into two types, *unidirectional* and *bidirectional.* In the unidirectional category (e.g., [4]) there is only one-direction type of process, emanating from the source node S and seeking the goal node G. The bidirectional category (e.g., [5, 6, 7, and 8]) incorporates two types of processes – one forward type search process, from S to G, and one reverse type search process, from G to S. Each of the processes can be executed using a typical unidirectional algorithm such as A* [4]. In some bidirectional search algorithms, special kind of nodes, called X-nodes (or islands), are used to aid the search. The concept of introducing X-nodes to speed up the

search was introduced in [9] and there have since been developed several algorithms that utilize X-nodes. Notable examples are PBA*, WS_PBA*, and SSC_PBA*, described in [7, 8].

Bidirectional multiprocessor heuristic search has been introduced by Nelson [13], first for 2 processors as a parallelization of Pohl's algorithm [5], and it was later generalized for N processors, N > 2 [11, 12]. The N-processor algorithm, PBA*, uses intermediate nodes of the search space, called *islands* or *X-nodes*. The terms *islands* and *X-nodes* are equivalent for the purposes of this paper. Historically, the term *island* nodes first appeared in [9] to denote intermediate nodes in the search space, with the property that an optimal path passes through at least one of them. The term *X-nodes* was introduced in [11] to denote intermediate nodes in the search space but not requiring that an optimal path passes through at least one of them. Also, since the algorithm described in [11] requires two search processes (one forward and one reverse) to emanate from each node, the term *X-node* was coined as a reminder of the bidirectionalism in the search (see figure 1) in order to divide the search space into smaller subspaces.

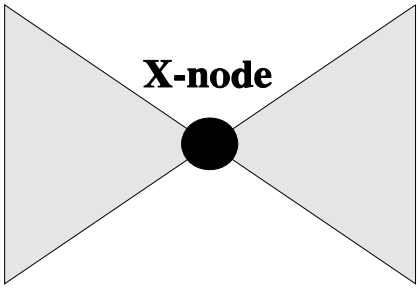

Figure 1. X-node.

In this paper, the terms *islands* and *X-nodes* are used interchangeably, and mean intermediate nodes of the search space, some of which may participate in a path (not necessarily optimal) connecting the source and goal nodes. Note, some of these intermediate nodes may not participate in a solution (actually this is especially true in [9] where only one of the island nodes participates in the solution). As illustrated in Figure 2, in addition to the parallel searches conducted

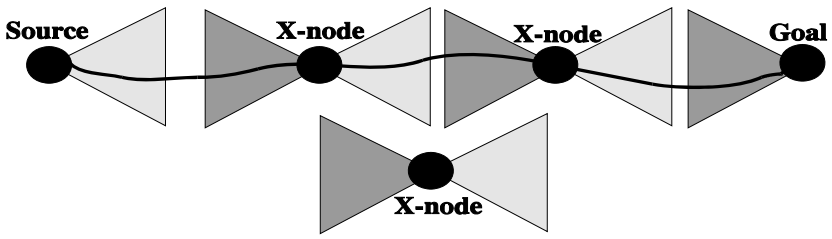

Figure 2. Parallel bidirectional search with islands.

from the source node S and the goal node G, two parallel searches are conducted from each X-node; a *forward* search towards the goal node G, and a *reverse* search towards the source node S. A solution is found as soon as a set of search spaces intersect in such a way that a path can be traced from S to G. The complexity of PBA* was analyzed in [11, 12]. In the case

that the X-nodes are placed equidistantly on an optimal path from source to goal, the complexity of PBA* is

$$O\left(b^{\frac{n}{2\cdot(|X|+1)}}\right)$$

where b is the branching factor of the problem (branching factor is the number of nodes generated at any node expansion), n is the length of an optimal path from source to goal, and |X| is the number of X-nodes (or islands). This situation is illustrated in Figure 3 for |X| = 2.

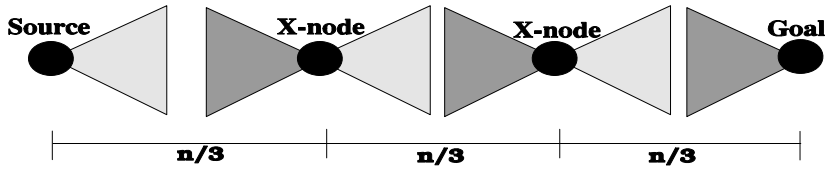

Figure 3. Algorithm PBA* in the best case.

As it is shown in Proposition 1 and Proposition 2 below, the speedup of PBA* over its uniprocessor version is potentially *linear on the number of processors*, and the speedup of PBA* over the conventional uniprocessor A* algorithm is potentially *exponential on the number of processors*. By "uniprocessor version of PBA*" it is meant that algorithm PBA* is simulated on a uniprocessor machine. This can be done, for example, by using a simulating program of a parallel machine running on a uniprocessor (such programs may be provided by the manufacturer of the parallel computer and are usually used for software development prior to porting to the actual parallel machine), or by simply spawning processes in a multitasking operating system like UNIX, or by utilizing the multithreading facility of modern programming languages and operating systems.

Proposition 1.
If all X-nodes are placed equidistantly on the straight line segment connecting the source and goal nodes, and the work is divided evenly among all the available processors, then the speedup of PBA* over UNI_PBA* is *linear* on the number of processors, assuming that each processor executes one search process of the PBA* algorithm.

Proof

$$Cost\ of\ PBA^* = O\left(b^{\frac{n}{2\cdot(|X|+1)}}\right).$$

$$Cost\ of\ UNI\_PBA^* = O\left(2\cdot(|X|+1)\cdot b^{\frac{n}{2\cdot(|X|+1)}}\right).$$

Therefore,

$$Speedup = \frac{Cost\ of\ UNI\_PBA^*}{Cost\ of\ PBA^*} = O\left(\frac{2\cdot(|X|+1)\cdot b^{\frac{n}{2\cdot(|X|+1)}}}{b^{\frac{n}{2\cdot(|X|+1)}}}\right) = O\left(2\cdot(|X|+1)\right).$$

Since $|P| = 2 \cdot (|X| + 1)$, where |P| is the number of processes running on either the uniprocessor or the multiprocessor machine, the above expression for the speedup becomes

$$Speedup = \frac{Cost\ of\ UNI\_PBA^*}{Cost\ of\ PBA^*} = O(|P|).$$

The result follows since one process of the PBA* runs on each processor.

Proposition 2.
If all X-nodes are placed equidistantly on the straight line segment connecting the source and goal nodes, then the speedup of algorithm PBA* over algorithm A* is *exponential* on the number of processors, assuming that each processor executes one search process of the PBA* algorithm.

Proof

$$Cost\ of\ PBA^* = O\left(b^{\frac{n}{2\cdot(|X|+1)}}\right).$$

$$Cost\ of\ A^* = O\left(b^n\right).$$

Therefore,

$$Speedup = \frac{Cost\ of\ A^*}{Cost\ of\ PBA^*} = O\left(\frac{b^n}{b^{\frac{n}{2\cdot(|X|+1)}}}\right) = O\left(b^{\frac{n\cdot(2\cdot|X|+1)}{2\cdot(|X|+1)}}\right) \qquad (1)$$

Since $|P| = 2 \cdot (|X| + 1)$, where |P| is the number of processes running on the multiprocessor machine (or simulated on the uniprocessor), expression (1) can be written as

$$Speedup = \frac{Cost\ of\ A^*}{Cost\ of\ PBA^*} = O\left(b^{\frac{n\cdot(|P|-1)}{|P|}}\right) \qquad (2)$$

Note, we can always write $n = k \cdot |P|$, for some *positive* number k (k is not necessarily an integer, but it is positive). Thus, expression (2) becomes

$$Speedup = \frac{Cost\ of\ A^*}{Cost\ of\ PBA^*} = O\left(b^{k\cdot(|P|-1)}\right) \qquad (3)$$

Expression (3) clearly shows that the obtained speedup of PBA* over A* is exponential on the number of processes, |P|. Note, this is the case even if k is less than 1 (but always bigger than 0). Since one process of PBA* runs on each processor, the obtained speedup is certainly exponential on the number of processors as well.

In the general case where X-nodes are not placed equidistantly on an optimal path from source to goal, the complexity of PBA* is

$$O\left(b^{\frac{\max\left(dist(X_i,X_j),dist(X_i,E)\right)}{2}}\right)$$

where X_i and X_j range over all X-nodes, and E is either the source or the goal node. This situation is illustrated in Figure 4 for two X-nodes. There are two main issues regarding the efficiency of PBA*. The first is search space management, that is, how to efficiently detect intersections among the many search spaces. The second is the X-node generation, that is, how to find X-nodes on a path from source to goal.

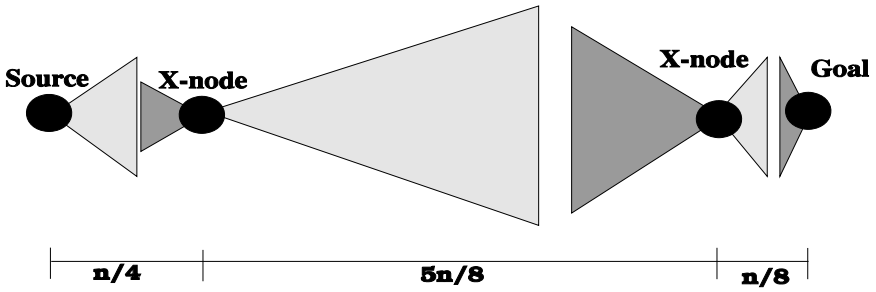

Figure 4. Algorithm PBA* in a general case.

In the case of search space management, the difficulty arises from the fact that when N searches are concurrently in progress, it is very time consuming to have each process test if one of its generated nodes is identical to some node generated by any other process (note, in the 2-processor case, checking for common nodes is fairly inexpensive, since there are only two searches in progress). As an attempt to ease this task, *multi-dimensional heuristics* (MDH) were developed by [11, 12]. As shown in Figure 5, additional nodes, called *reference* nodes, are designated into the search space, in order to "guide" the search. In Figure 5, $k_1 = k_2 < k_3$ and $d_1 = d_2 = d_3$. This leads to the prediction that the search spaces S_1 and S_2 are most likely to intersect, while search space S_3 does not. It has been shown in [11, 12] that if two search spaces S_1 and S_2 intersect, then there exist nodes $n_1 \in S_1$ and $n_2 \in S_2$ such that

$$dist(n_1, R_k) = dist(n_2, R_k)$$

for all reference nodes R_k. Actually, nodes n_1 and n_2 represent the same state in the global state space. This state is common to S_1 and S_2. Note, the reverse statement may not be true, since, for example, two diametrically opposite nodes also satisfy this condition although they do not lie in the same search space.

Based on the reference nodes use in MDH, several *intersection detection algorithms* have been devised [11, 10, and 12]. The algorithms' task is to prevent node exchange between any two processes, unless there is a high probability that the search spaces of these processes intersect. Simulation results have shown that in the worst case, unnecessary node exchange is cut down in half, whereas in the best case it is cut down almost completely (only 1% of the nodes are exchanged between two non-intersecting subspaces). Later in this paper we outline some of the intersection detection algorithms since we use them in the implementation of our proposed method here.

2. Proposed Methods

In this section we present two algorithms that address the issues outlined in the previous section. The first issue is how to select X-nodes (island nodes) which are located appropriately in the state space. Algorithm A, later in this section, describes the proposed

method. The second issue is how to select R-nodes (reference nodes) which are located appropriately in the state space. Algorithm B, later in this section, describes the proposed method.

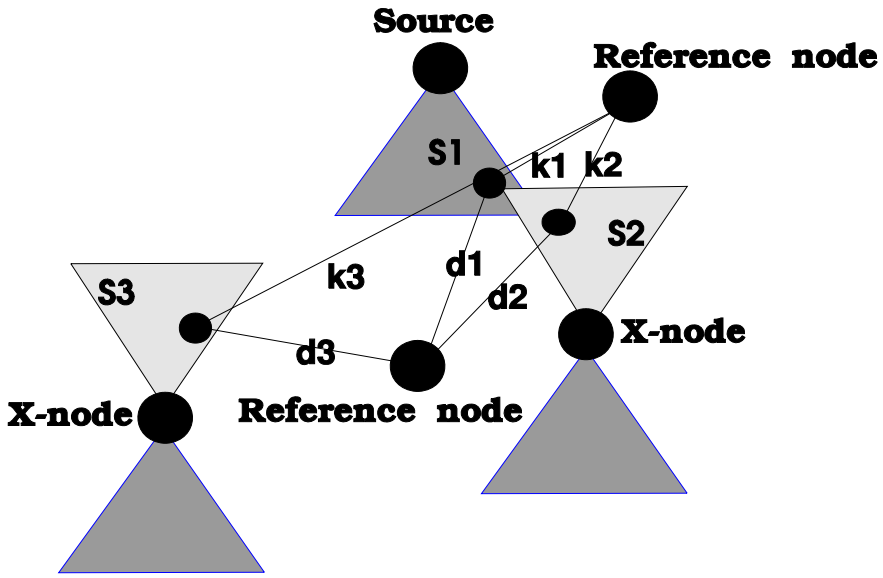

Figure 5. Reference nodes in PBA*.

Note, although the above two issues are both within the same algorithm, PBA*, they represent different problems. For the first issue (X-node placement), the purpose is to generate X-nodes that are located (hopefully) on, or close to, a path connecting nodes S and G. The motivation behind this problem is that during the execution of PBA* a path can be formed easier (faster) if appropriate X-nodes have been selected. For the second issue (R-node placement) the purpose is to generate R-nodes that are located far away from each other so that they can provide a better guide for the multidimensional heuristics component of PBA*. Each of the two methods (generating appropriate X-nodes and generating appropriate R-nodes) can be used independently to improve the overall performance of PBA*, although the two methods could be also combined, to incur a cumulative effect in the improvement of PBA*.

We use two different tools to address the above two separate issues. For the X-nodes placement issue, we employ Voronoi diagrams. For the R-nodes placement issue we employ Kohonen neural networks.

Voronoi diagrams [23, 22, and 24] are named after the Russian Mathematician Georgy F. Voronoi [21]. They are geometric structures that partition a N-dimensional space into cells such that the points within each cell are closer to a *centroid* point of that cell than to any other point of any other cell. Figure 6 shows a Voronoi diagram in 2-dimensional space, created by using an applet in [17].

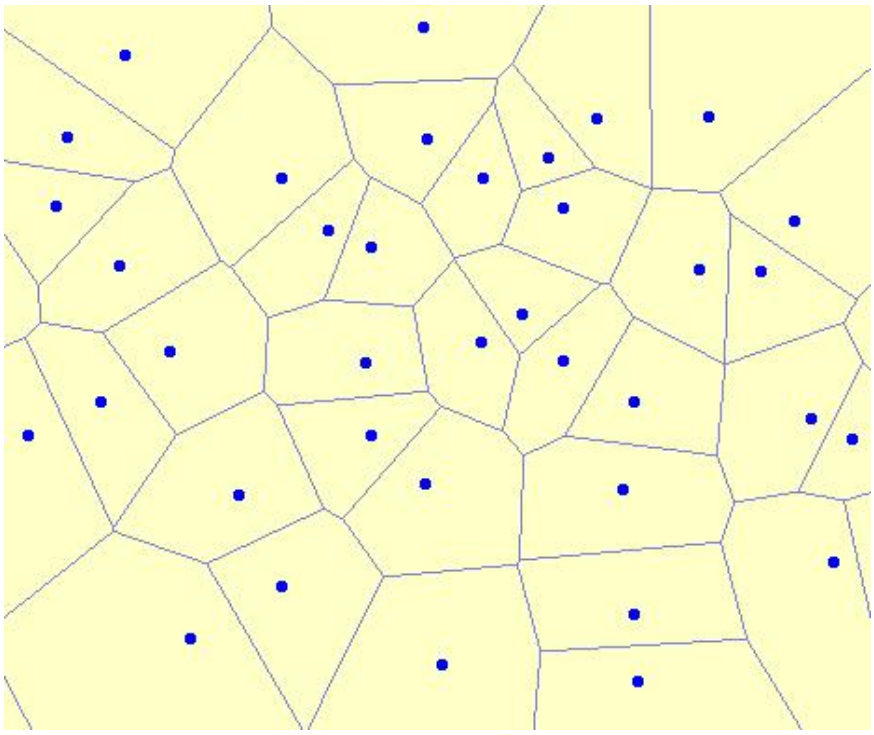

Figure 6. Sample Voronoi diagram.

The points (dots) in Figure 6 are the centroids of the corresponding Voronoi cells. As can be seen from Figure 6, the main characteristic of Voronoi diagrams is that any point within a Voronoi cell VC is closer to the centroid of VC than to any point of any other cell. In this sense, a Voronoi diagram partitions a space in such a way that all close-to-each-other points are clustered into cells such that each cell has a representative point – the centroid of that cell, serving as the proximity center of the region of space covered by that cell. This characteristic of Voronoi cells – especially the existence of the centroids is what inspired us to adopt the use of Voronoi diagrams to address the issue of determining appropriately located X-nodes in a state space. Specifically, the idea is to partition the state space into Voronoi cells and then to designate as X-nodes *some* of the centroids of the generated cells. The benefits of such an approach could be two-fold. First, by carefully selecting which Voronoi cells will participate in the solution, we *avoid* selecting *irrelevant* centroids (this would lead to X-nodes that are obviously *out of any* path connecting S and G). Second, and most important, by designating as X-nodes the centroids of *neighboring cells that form a path* connecting S and G, it is hoped that the centroids of those cells will lead to forming a path connecting S and G. Note, the dual graph of any Voronoi diagram is the Delaunay graph [24]. Therefore, if neighboring Voronoi cells form a path connecting S and G, then the corresponding vertices of the Delaunay graph of that Voronoi diagram form a path connecting S and G! Algorithm A below, describes the above process in detail.

Algorithm A.

- *Step A.1*: generate a set $X = \{x_1, ..., x_M\}$ of *candidate* X-nodes.

- *Step A.2*: convert set X into set $P = \{\pi_1, ..., \pi_M\}$ such that π_i is the projection of x_i onto a 2-dim space.
- *Step A.3*: Form D-graph, the Delaunay graph for set P.
- *Step A.4*: Calculate the *shortest* path SP = [S-X1-X2- …-Xk-G] connecting S and G in the D-graph of step A.3.
- *Step A.5*: Select as X-nodes the nodes corresponding to points X1, X2, …, Xk and execute algorithm PBA* between S and G.

In *step A.1*, we randomly generate a fairly large number of nodes. These are *candidate* X-nodes and the intention is that after the processing done in steps A.2, A.3, and A.4, a small number of those nodes is selected for being used in algorithm PBA*. In *step A.2*, we map the nodes x_i generated in step A.1 onto points π_i of an N-dim space. In our case, we use N=2. (This is because a 2-dim space is more convenient to use for step A.3. In the future we plan to extend our work for N > 2). To project a node x_i onto a point π_i in 2-dim space we use two randomly generated nodes r_1 and r_2 (other than the ones of set X of step A.1). Then π_i is calculated as

$$\pi_i = \left(md\left(x_i, r_1\right), md\left(x_i, r_2\right) \right)$$

where $md\left(x_i, r_j\right)$ is the Manhattan distance between x_i and r_j (j=1, 2). In *step A.3*, we form the D-graph of the set P calculated in step A.2. This graph is the Delaunay triangulation DT(P) for the set of points P. The main property of DT(P) is that no point in P is inside the circumcircle of any triangle of DT(P). Also, the DT(P) corresponds to the dual graph of the Voronoi diagram for P. This latter property is our main motive in calculating the DT(P). Without loss of generality, we assume that the vertices of DT(P) correspond to the centers/generators of the Voronoi cells in the Voronoi diagram that corresponds to DT(P). The property of interest from any Voronoi diagram is that all the points within a Voronoi cell are closer to the center of that cell than to the center of any other cell, including the neighboring cells. Therefore, by considering the Voronoi diagram of the DT(P) graph, we somewhat avoid choosing as X-nodes nodes that are too close to each other. (Note, such X-nodes would correspond to search spaces that possibly duplicate each other and thus waste resources by exploring the same part of the global search space). Consequently, we designate as our X-nodes the generator nodes of the Voronoi cells that form a *shortest* path between S and G. In *step A.4*, we apply Dijkstra's shortest path algorithm on graph DT(P) and calculate SP = [S-X1-X2- …-Xk-G], the shortest path connecting S and G. Note, S and G of path SP of step A.4 are not the actual source and goal *nodes* but they are the 2-dim *points* (as calculated in step A.2) corresponding to the actual source and goal nodes. Also, X1, …, Xk of step A.4 are 2-dim points corresponding to certain candidate X-nodes generated during step A.1. For simplicity in notation we use the same symbols to refer to the corresponding actual nodes S and G and the corresponding X-nodes of the 2-dim points X1, …, Xk. In *step A.5*, we designate as our X-nodes the nodes that correspond to the points X1, …, Xk, and using those nodes we execute algorithm PBA* between S and G.

Kohonen Nets [19, 18] (also known as Kohonen Neural Networks, or Kohonen Maps, or Self-Organizing Maps (SOM)) are named after Finnish Professor Teuvo Kohonen. A Kohonen Net (KHNN) is typically a 2-dimensional grid of nodes with the ability to organize these nodes in such a way that similar nodes end-up close to each other in the grid. Figure 7 illustrates such a grid of nodes.

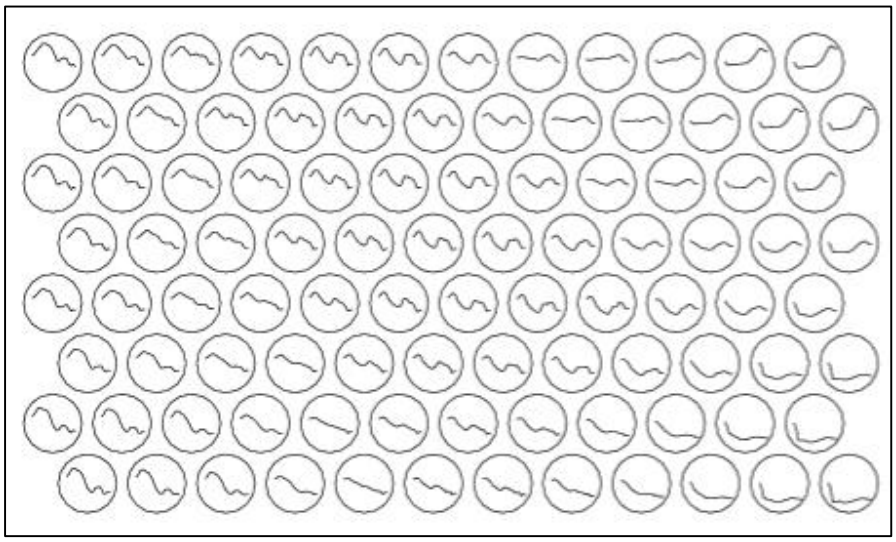

Figure 7. A Kohonen map [20].

As we observe in Figure 7, similar symbols tend to cluster together (e.g., the symbols at the upper-left region of Figure 7 are very similar). This characteristic of Kohonen nets is what inspired us to adopt the use of Kohonen nets for addressing the issue of determining appropriate R-nodes in a state space. Note, it is desirable that the R-nodes are well-distributed in the state space, prior to the start of algorithm PBA*. The idea is that for a large number of randomly generated nodes, a Kohonen net will cluster those nodes in such a way that similar nodes are placed close to each other in the SOM. Then, we choose as our R-nodes from the final SOM configuration *no more than one* node from each such cluster. The idea is that, hopefully, such a selection will lead to selected R-nodes which are "far away" from each other in the state space. The benefit of this is the increase of the ability of algorithm PBA* to understand when two nodes that belong to opposing search processes are really close to each other, or, may be, even identical. This undersraning is important because, during the execution of PBA*, if two opposing search processes S1 and S2 generate nodes N1 and N2 such that N1 is similar to N2, then S1 and S2 have high probability to intersect, i.e., contain a common node and, thus, form a path connecting the X-node X1 (origin of S1) and X-node X2 (origin of S2), and, obviously, if such partial paths are found throughout the opposing search spaces that execute (concurrently) between nodes S and G, then a (whole) path is formed between S and G – and this path would signal the completion of PBA*. If the R-nodes that are used in PBA* are well-spread out in the state space, then the estimates of when two nodes N1 and N2 are close to each other tend to be more reliable than if the R-nodes are not well-spread out in the state space. This scenario is illustrated in Figure 8.

In Figure 8, nodes N1 and N2 (of opposing search processes) are indeed close to each other and this closeness is predicted faithfully by calculating the corresponding distances to the R-nodes R1, R2, and R3. Note, for those distances $D11 \approx D21$ and $D12 \approx D22$ and $D13 \approx D23$. That is, $(D11, D12, D13) \approx (D21, D22, D23)$, and this leads to the (correct) prediction that $N1 \approx N2$, i.e., that nodes N1 and N2 are either identical, or very similar.

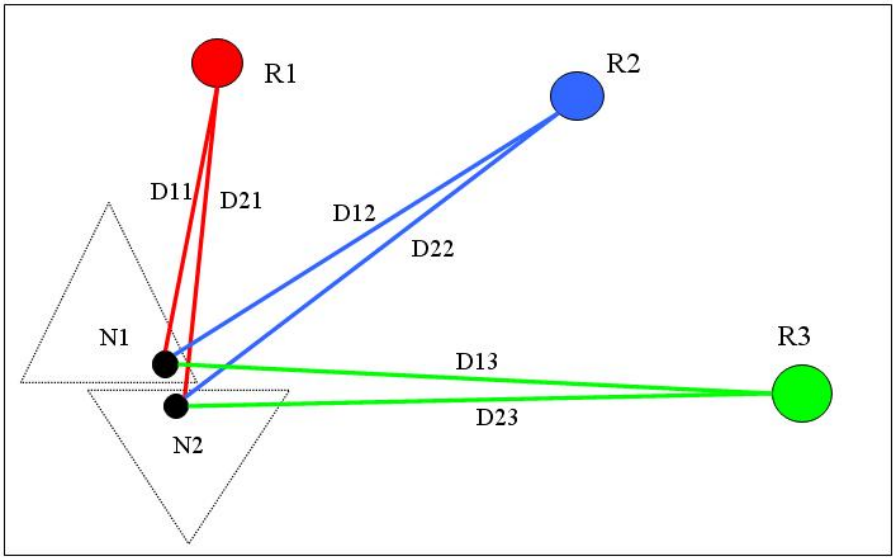

Figure 8. R-nodes far-apart in the state space.

The implication of such a prediction is that the opposing search processes P1 and P2 that generated nodes N1 and N2, respectively, will initiate communication for the purpose of finding out if indeed their search spaces S1 and S2 contain a common node (such a finding would mean that a path connecting the origin node of S1 and the origin node of S2, has been established. However, if the R-nodes are located very close to each other, the above test for node closeness may lead to a faulty prediction, as illustrated in Figure 9.

In Figure 9, nodes N1 and N2 (of opposing search processes) are actually far apart from each other. As such, based on the non-closeness of N1 and N2, the system should not instruct the corresponding processes P1 and P2 to initiate communication. Nevertheless, because the R-nodes R1, R2, and R3 are close to each other, $D11 \approx D12 \approx D13$, and $D21 \approx D22 \approx D23$, and $D31 \approx D32 \approx D33$. These similarities alone do not lead to a prediction that nodes N1 and N2 are close to each other; however, due to a coincidental placement of N1 and N2 as shown in Figure 9, it is also $D11 \approx D21$ and $D12 \approx D22$ and $D13 \approx D23$, which then definitely leads to the faulty prediction that $N1 \approx N2$! Then, like in the case of the correct prediction (of Figure 8) but, in this case, under erroneous conditions, the system will initiate communication between search spaces S1 and S2 for the purpose of trying to determine if S1 and S2 contain a common node. Note, most likely, such communication will yield no common node, and would rather waste communication bandwidth. Interestingly, if another R-node, RX, is added in the state space, with RX far away from R1, R2, and R3, the above situation will not arise! This is illustrated in Figure 10.

In Figure 10, the above anomaly is still there for R-nodes R1, R2, and R3, but the fourth R-node RX breaks the tie since the distances D1X and D2X are very different.

Based on the above discussion, we use a Kohonen Net to generate appropriately placed R-nodes, as follows. First, we generate *many* nodes, randomly. These are candidate R-nodes. We place those nodes on a (unorganized) Kohonen map. Then we execute Kohonen's self-organizing algorithm on that map.

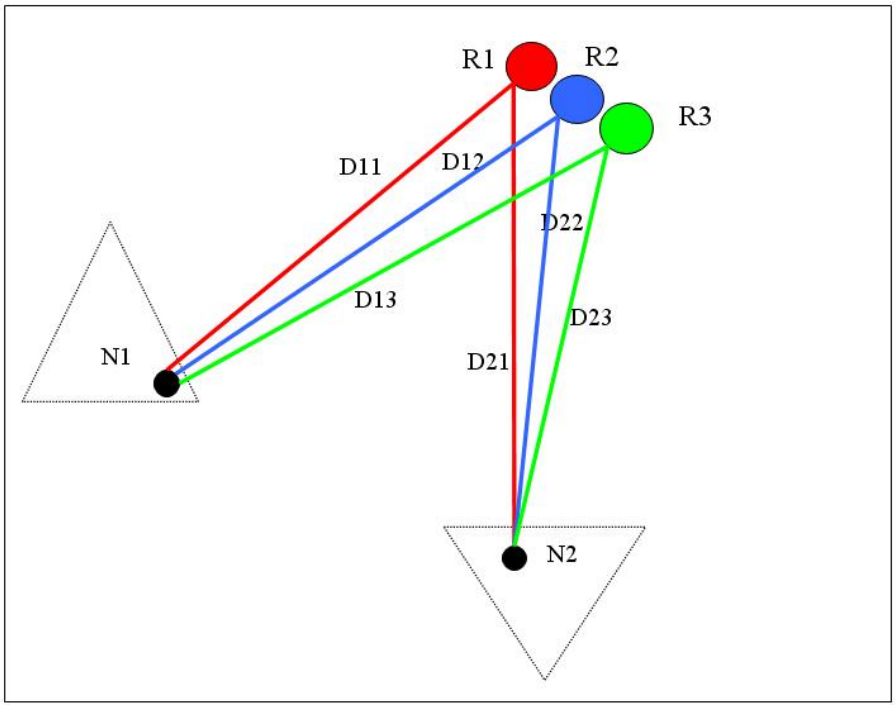

Figure 9. R-nodes close to each other in the state space.

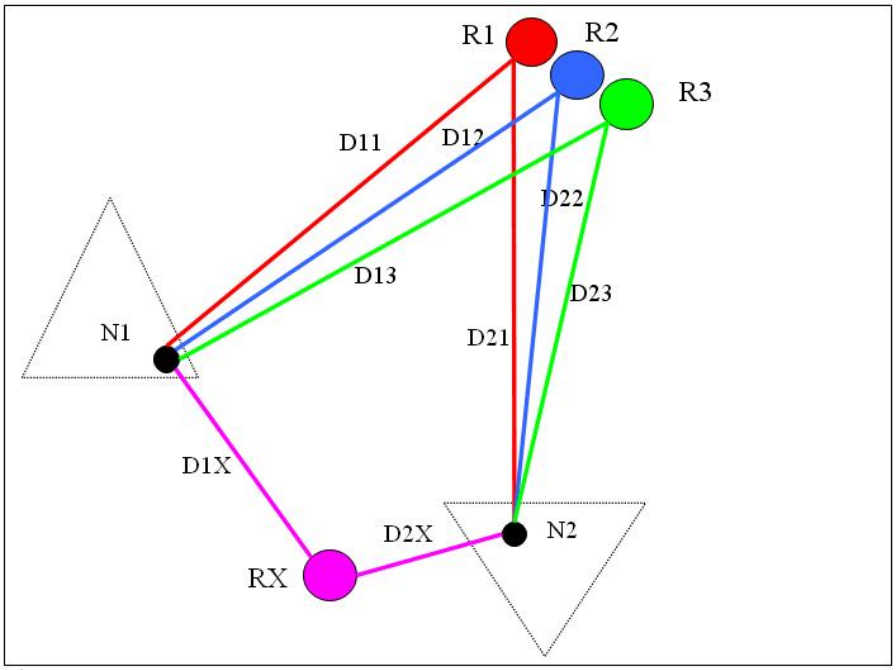

Figure 10. R-node RX resolves closeness confusion of N1 and N2.

The result is a self-organized Kohonen grid with the characteristic that closely located nodes on that grid are similar (along the lines of the Kohonen map of Figure 7). Then, we select as our R-nodes nodes from the organized Kohonen map that are far away from each other. It is hoped that by doing so, the selected nodes to be used as our R-nodes for algorithm PBA* are not similar to each other and, therefore, do not incur the R-node closeness anomaly discussed earlier (and illustrated in Figure 9).

Algorithm B, next, describes the above process.

Algorithm B.

- *Step B.1*: Generate a set $R = \{r_1, ..., r_N\}$ of *candidate* R-nodes, and initialize a Kohonen Map with the nodes of set R.
- *Step B.2*: Execute Kohonen's algorithm and self-organize the map of Step B.1.
- *Step B.3*: Select a desired number of nodes from the organized map of step B.2, such that the selected nodes are located far away from each other on the map.
- *Step B.4*: Generate, randomly, a desired number of nodes to be used as X-nodes.
- *Step B.5*: Execute algorithm PBA* between S and G, using as R-nodes the nodes selected in step B.3 and as X-nodes the nodes generated in step B.4.

In *step B.1* of algorithm B, we randomly generate a fairly large number of nodes. Those are candidate R-nodes and the intention is that after the processing done in steps B.2 and B.3, a small number of nodes among those nodes are selected for being used as R-nodes in algorithm PBA*. Prior to beginning step B.2, the random nodes generated in step B.1 are placed on a Kohonen Map. In *step B.2*, the Kohonen map initialized in step B.1, is self-organized. This is done by executing Kohonen's algorithm for SOMs. In step B.3, we select a few nodes from the organized map and use them as R-nodes for PBA*.

3. Performance evaluation

We implement A*, the traditional heuristic search algorithm, and three versions of PBA*: PBA*-R, PBA*-VD, and PBA*-KH. PBA*-R is PBA* with randomly generated X-nodes and R-nodes; PBA*-VD is PBA* with Voronoi-Dijkstra designated X-nodes, per Algorithm A of Section 2, and randomly generated R-nodes; PBA*-KH is PBA* with Kohonen designated R-nodes, per Algorithm B of Section 2, and randomly generated X-nodes. We use the sliding tiles puzzle problem for our tests and run A*, PBA*-R, PBA*-VD, and PBA*-KH for a variety of puzzles. In comparing PBA*-R versus PBA*-VD, we use 17 puzzles (3 of the puzzles are 6x6, i.e., 35-puzzles, 4 are 5x5, and 10 are 4x4 puzzles). In comparing PBA*-R versus PBA*-KH, we use 13 4x4 puzzles (we tested the algorithms for several larger puzzles and for additional 4x4 puzzles, but algorithm PBA*-R did not complete its execution for those, and the results are not reported in this paper). We use the Manhattan distance as our heuristic function. Algorithms PBA*-R and PBA*-VD complete their execution and find solutions for all 17 puzzles. Due to memory space limitations, algorithm A* is not able to find a solution for any of the 6x6 puzzles, and for 2 of the 5x5 puzzles. Although algorithms PBA*-R, PBA*-VD, and PBA*-KH are not admissible, they find near-optimal solutions. For either version of PBA*, an *intersection detection algorithm*, IDA-3, is used to control search process communication. IDA-3 is originally described in [10] and it has been used in several of our previous works (e.g., [7, 8, 14, 15, 16]). The main characteristic of IDA-3 is that it instructs two opposite-direction search processes to exchange nodes (and, henceforth, compare those nodes) if such nodes are deemed to be "similar

enough" so that they are possibly identical. Since algorithm IDA-3 is central in our evaluation due to its communication cost, we describe IDA-3 here.

Intersection Detection Algorithm IDA-3.

Every search space is represented as a N-dimensional polyhedron with 2^N corners. Each corner has coordinates

$$\left(R_{1i}, R_{2i}, ..., R_{Ni}\right)$$

where R_{ji} is either the minimum or the maximum distance of the X-node corresponding to that search space, from the reference node R_j, j = 1, ..., N. Two search spaces S_a and S_b may contain a common node when there is an overlap of their corresponding polyhedra. A common node between two search spaces is not possible to exist, until such an overlap occurs.

Two search spaces S_a and S_b might contain an intersection when there is an overlap for each of their reference node ranges. Pictorially this is represented by an intersection of the approximated spaces in the N-space. In Figure 11 this occurs for S_3 and S_4 (for 2 reference nodes).

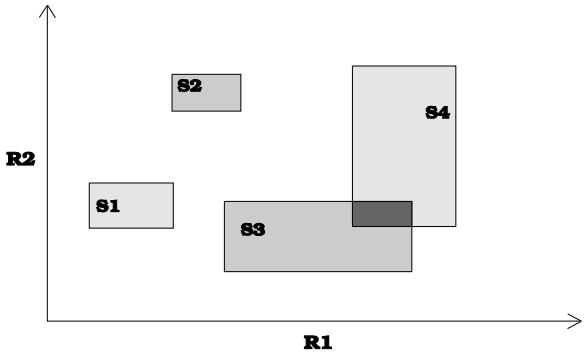

Figure 11. N-space intersection in PBA* (N = 2).

In algorithm PBA* a *central control process* (CCP) is used to coordinate the local search processes running concurrently. The CCP compares the search spaces with respect to their estimated distances to the reference nodes and instructs two search processes to start exchanging nodes when their search spaces seem to be intersecting. As it is quite expensive to store *all* the reference nodes' distances for *all* nodes generated by each search space, the CCP uses an *overlap table* to store selected distances for each reference node. Specifically, the overlap table holds only the minimum and maximum distances to each reference node from each search space. Each search space keeps track of its minimum and maximum values for each reference node, and informs the CCP as these values change. The CCP receives the new values, updates the overlap table, and checks if an overlap between a pair of opposite direction search spaces has occurred. An overlap with respect to a reference node k occurs when

$$\left(S_i R_k \min, S_i R_k \max\right) \cap \left(S_j R_k \min, S_j R_k \max\right) \neq \varnothing$$

for **all** reference nodes R_k, where

$$S_j R_k \min = \min\left\{dist\left(n_{S_j}, R_k\right)\right\}$$

and

$$S_j R_k \max = \max\left\{dist\left(n_{S_j}, R_k\right)\right\}$$

for any node n_{S_j} in S_j.

It happens that an intersection between two search spaces is not possible until such an overlap occurs. Figure 12 is an example of an overlap table maintained by the central control process when four search processes and four reference nodes are present. The table shows that search processes S_1 (forward) and S_2 (reverse) overlap with respect to all four reference nodes. In this case, the CCP will instruct search processes S_1 and S_2 to start node exchange. In particular, the reverse search process S_2 is instructed to send nodes to the forward search process S_1, and the forward search process S_1 is given an alert that nodes from S_2 are to arrive.

Search process	**R1**		**R2**		**R3**		**R4**	
	Min	Max	Min	Max	Min	Max	Min	**Max**
S1 (forward)	10	17	63	68	4	25	9	**26**
S2 (reverse)	14	27	43	65	13	31	18	**40**
S3 (forward)	4	7	11	19	4	12	4	**19**
S4 (reverse)	**28**	**32**	**22**	**31**	**37**	**51**	**28**	**43**

Figure 12. A sample overlap table.

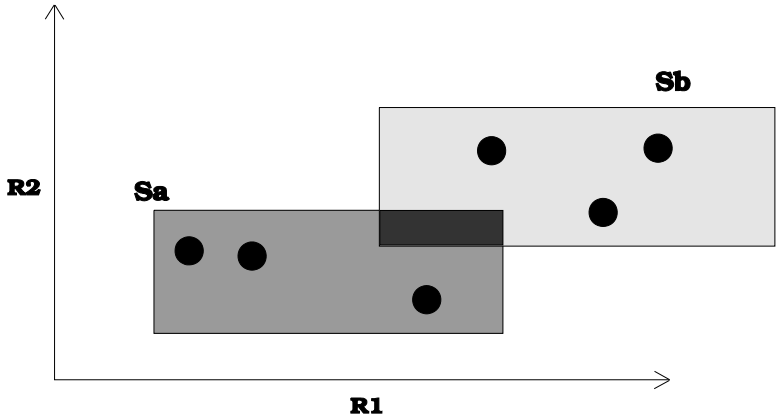

Figure 13. S_a and S_b overlap, but $S_a \cap S_b = \varnothing$.

If the central control process uses an overlap table to decide when to turn on communication between two searches, it can be assured that some existing intersection will not be overlooked. Unfortunately, the existence of these overlaps does not insure that the two search spaces contain a common node. The condition is not sufficient for several reasons, one of which is that the heuristics only *estimate* distances to the reference nodes. However, even if the heuristic makes no errors in estimating distances, there are still other reasons why this condition is not sufficient. As shown in Figure 13, it may be that the approximated space of

S_a intersects with the approximated space of S_b without having any single node in (S_b) S_a overlap all the reference node ranges in (S_a) S_b. The result of this is that no nodes lie in the intersection of the approximated spaces [11]. Three different intersection detection algorithms, which increasingly progressive degree of refinement, are outlined next.

(1) *IDA-1* maintains a [min, max] overlap table and when the approximated spaces of S_f and S_r intersect, S_r will begin sending its nodes to S_f.

(2) *IDA-2* is based on addressing one of the reasons which keeps the overlap condition, the basis of IDA-1, from being sufficient to insure that two spaces intersect. IDA-2 waits until the reference node values of a single node in S_r lie in the approximated space of S_f. When this condition occurs, S_r will then begin sending its nodes to S_f.

(3) *IDA-3* checks for the same condition as IDA-2 except that it does so on a node by node basis. Both IDA-1 and IDA-2 end up turning on communication permanently. It may be that two search spaces only come close to intersecting for a short period of time resulting to a lot of unnecessary node sending if communication is permanently turned on. Instead, IDA-3 checks as each node is generated and only sends nodes which lie in the approximated space of S_f.

As mentioned earlier, we adopt IDA-3 for our evaluation. Our experiments reveal the following.

Result 1: For the vast majority of tests (more than 80% of test cases), the X-nodes used in algorithm PBA*-VD are more likely to aid in establishing a path connecting S and G than the X-nodes used by algorithm PBA*-R. This is illustrated by Table 1.

Table 1. IDA-3 probes for possible search space intersection.

	PBA*-R	**PBA*-VD**	**Winner**
	16,642	25,937	**PBA*-VD**
	15,573	38,015	**PBA*-VD**
	27,980	27,462	**PBA*-R**
	35,753	43,784	**PBA*-VD**
	1,106	3,380	**PBA*-VD**
	28,756	29,584	**PBA*-VD**
	56,574	72,625	**PBA*-VD**
	15,605	19,599	**PBA*-VD**
	239,480	231,296	**PBA*-R**
	61,741	73,515	**PBA*-VD**
	28,598	31,534	**PBA*-VD**
	227,819	235,092	**PBA*-VD**
	150,297	153,509	**PBA*-VD**
	48,063	58,671	**PBA*-VD**
	165,146	205,781	**PBA*-VD**
	69,090	85,686	**PBA*-VD**
	143,781	132,448	**PBA*-R**
Total	1,332,004	1,467,918	
Average	78,353	86,348	
wins of PBA*-VD over PBA *-R			**82.35%**

Note, for all but three cases in Table 1, algorithm PBA*-VD probes more times for search space intersection.

Result 2: The overhead for incorporating the Voronoi-Dijkstra method in finding useful X-nodes is *negligible*. This is illustrated by Table 2.

Table 2. Total overhead (over all test puzzles).

method	Total (sec)
PBA*-R	**6,435**
PBA*-VD	**6,103**
GRN overhead	**16.437**
D/D overhead	**8.819**
Total Overhead (GRN + D/D)	**25**

As shown in Table 2, there are two types of overhead in incorporating Algorithm A of section 2 into algorithm PBA*: the *GRN overhead*, and the *D/D overhead*. The GRN overhead is the cost of executing essentially steps A.1 and A.2 of Algorithm A (generate candidate X-nodes and project them onto a 2-dim space). The D/D overhead is the cost of executing steps A.3 and A.4 of Algorithm A (form the D-graph and calculate shortest path with Dijkstra's algorithm). As we see in Table 2, the *total overhead* (GRN + D/D) is 25 seconds, which is 0.41% (25/6103) of the time required to execute algorithm PBA*-VD. We also note that the total time to execute PBA*-VD (i.e. the time for the actual PBA*-VD plus the total overhead time) does not exceed the time to execute PBA*-R. Therefore, assuming that the incorporation of the Voronoi-Dijkstra technique into algorithm PBA* does not, in any way, harm the overall quality of the heuristic search process, the overhead for employing the Voronoi-Dijkstra method for finding useful X-nodes is not only negligible, but it also positively contributes (as evidenced by the results shown in Table 1), to the overall quality of PBA*.

Result 3: *The Voronoi-Dijkstra "anomaly"*. Our experiments uncover an unfortunate scenario, illustrated in Figure 14.

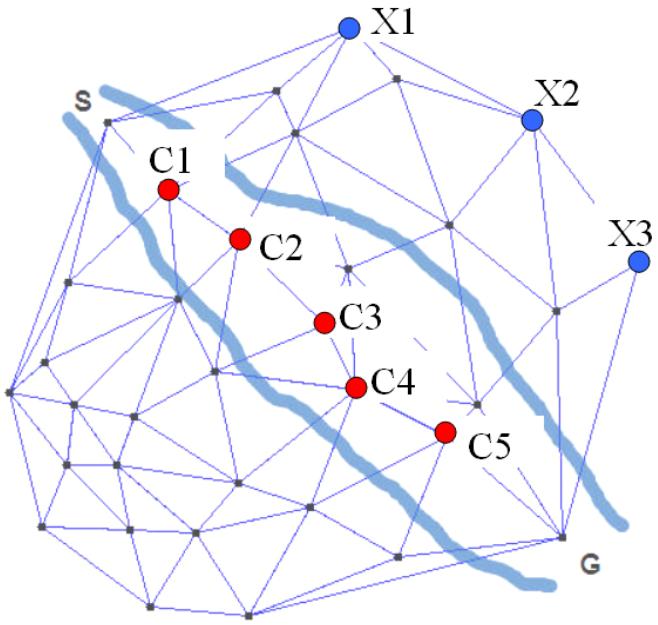

Figure 14. The Voronoi-Dijkstra "anomaly".

Figure 14 shows the D-graph formed by a specific arrangement of randomly generated candidate X-nodes. The path S, X1, X2, X3, G is the shortest path connecting S and G and, therefore, X1, X2, and X3 are chosen as the X-nodes for the execution of algorithm PBA*-VD. Note, however, although these nodes are the ones that form a shortest path between S and G, nodes C1, …, C5 seem to be a better alternative for X-nodes to use for PBA*-VD! This leads us to speculate that the *shortest path* (as calculated by the Dijkstra algorithm) *may not be the best choice* for X-nodes and, instead, the *straightest path* between S and G might be a *better alternative*! Investigation of the ramifications of this "anomaly" is in our immediate research plans.

Result 4: For the vast majority of tests (nearly 77% of test cases), when comparing PBA*-R vs PBA*-KH, the R-nodes generated by the Kohonen method for algorithm PBA*-KH are more suitable to be used in PBA* than randomly generated R-nodes. This is illustrated by Table 3.

Table 3. PBA*-R vs PBA*-KH; IDA-3 probes.

	PBA*-R	PBA*-KH	Winner
	120,715	129,608	**PBA-R**
	398,695	390,862	**PBA-KH**
	133,128	58,617	**PBA-KH**
	101,938	87,605	**PBA-KH**
	128,790	116,907	**PBA-KH**
	399,206	262,215	**PBA-KH**
	140,389	97,290	**PBA-KH**

	103,313	107,407	**PBA-R**
	249,473	218,417	**PBA-KH**
	90,892	96,533	**PBA-R**
	115,640	106,108	**PBA-KH**
	85,774	80,993	**PBA-KH**
	256,751	96,741	**PBA-KH**
Total	**2,324,704**	**1,849,303**	
Average	**178,823**	**142,254**	
wins of PBA*-KH over PBA *-R			**76.92%**

Note, for all but three cases in Table 3, the number of probes incurred by algorithm PBA*-KH is less that the corresponding number for algorithm PBA*-R. This means that the Kohonen generated R-nodes serve as a better guide to predict opposing search space intersections and thus, communication of opposing search processes is triggered less often in PBA*-KH than in PBA*-R.

4. Conclusion

We present a method to generate appropriately located island nodes (X-nodes) and a method to find appropriately located reference nodes (R-nodes), within a search space. The motive for doing so for the X-nodes is that is that such generated nodes will help establish a solution path faster, if used by a multi-process bidirectional heuristic search algorithm, such as PBA*. The motive for doing so for the R-nodes is that appropriately located R-nodes provide a more accurate estimate of opposing search space intersections and thus help reduce interprocess communication cost in PBA*. To the best of our knowledge both of these problems have resisted any type of general purpose solution for more than two decades. We implement our methods and test them using PBA*, a bidirectional multi-process heuristic search algorithm designed to utilize X-nodes and R-nodes. Our findings indicate that PBA*-VD (the version of PBA* that uses the island nodes generated by our method) outperforms PBA*-R (the version of PBA* that uses randomly generated island nodes), more than 80% of the time. Also, the overhead of incorporating our method into PBA* is negligible (less than 0.5% of the cost of executing the PBA* algorithm itself). Interestingly, we also uncover an "anomaly" (Result 3, in Section 3), whose remedying points to a method of generating even more appropriately located island nodes. For the proposed method of locating appropriate R-nodes, our findings indicate that algorithm PBA*-KH (the version of PBA* that uses R-nodes generated by our method) outperforms PBA*-R more than 75% of the time. Also, like in the case of PBA*-VD, the cost of incorporating our method into PBA* is negligible.

Our future research plans include investigating the ramifications of the found "anomaly" for the case of PBA*-VD and also extending our method to N-dimensional spaces, when N > 2. For the case of PBA*-KH, there are several issues susceptible to improvement – especially during the execution of the Kohonen self-organizing algorithm. Note, the main ingredients of that algorithm are the *learning rate* (i.e., how much the winning node within the Kohonen grid should learn from the input vector) and the *neighborhood function* (which determines which neighbors of the winner node should learn from the input, and, in conjunction with the learning rate, it also determines how much those neighbors should learn form the input). In our implementation we use the number of moves made to the empty tile of a NxN puzzle and translate the learning rate and neighborhood values into that number of moves (note, unlike

many problems where Kohonen NN are used, the nature of our problem – and of many problems where heuristic search algorithms are applicable, is such that arithmetic operations between two Kohonen map neurons are meaningless). A side-effect of this policy is that *fractional* values are meaningless and, as such, are discarded from our calculations, by rounding the corresponding values where applicable. If we can somehow incorporate those lost fractional values into the number of moves, we believe that our results will be more accurate and PBA*-KH will produce better performance. Also, as an overall issue, we would like to amalgamate PBA*-VD and PBA*-KH into a single algorithm, and investigate the benefits of that approach. We hope and expect that such an algorithm will result in an overall performance that improves the currently focusing only to X-nodes, or only to R-nodes implementations of PBA*-VD and PBA*-KH, respectively. We are currently working toward investigating this approach.

References

[1] G. Antoniol, M. Di Penta, and M. Harman, "Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project", Proceedings of IEEE International Conference on Software Maintenance, 2005, pp. 240-249.
[2] F. Yu, H.S. Ip, and C.H. Leung, "A Heuristic Search for Relevant Images on the Web", Lecture Notes in Computer Science, Springer, Vol. 3568, 2005, pp. 599-608
[3] R. Bekkerman, S Zilberstein, and J. Allan, "Web Page Clustering using Heuristic Search in the Web Graph", Proceedings of IJCAI-07, the 20th International Joint Conference on Artificial Intelligence, 2007.
[4] P.E. Hart, N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems, Science, and Cybernetics, vol. 4, no. 2, 1968, pp. 100–107.
[5] I. Pohl, "Bi-Directional Search", Machine Intelligence, 1971, pp. 127-140.
[6] D. DeChampeaux, "Bidirectional Heuristic Search Again", Journal of the ACM, Vol. 30, No. 1, 1983, pp. 22-32.
[7] P.C. Nelson, and A.A. Toptsis, "Superlinear Speedup Using Bidirectionalism and Islands", Proc. International Joint Conference on Artificial Intelligence (IJCAI) – Workshop on Parallel Processing in AI, Sydney, Australia, 1991, pp. 129-134.
[8] P.C. Nelson, and A.A. Toptsis, "Unidirectional and Bidirectional Search Algorithms", IEEE Software, Vol. 9, No. 2, March 1992, pp. 77-83.
[9] P.P. Chakrabarti, S., Ghose, and S.C. Desarkar, Heuristic Search Through Islands, Artificial Intelligence, vol. 29, 1986, pp. 339-348.
[10] P.C. Nelson, and L. Henschen, "Multi-Dimensional Heuristic Searching", IJCAI '89 - International Joint Conf. on Artificial Intelligence, 1989, pp. 316-321.
[11] P.C. Nelson, "Parallel Bidirectional Search Using Multi - Dimensional Heuristics", Ph.D. Dissertation, Northwestern University, Evanston, Illinois, June 1998.
[12] P.C. Nelson, Parallel Heuristic Search Using Islands, *P*roc. 4th Conf. on Hypercubes, Concurrent Computers and Applications, Monterey, March 1989..
[13] P.C. Nelson, and L. Henschen, "Parallel Bidirectional Heuristic Searching", Proc. Canadian Information Processing Society 5, Montreal, Canada, 1987, pp. 117-124.
[14] A.A. Toptsis, and P.C. Nelson, "Parallel Bidirectional Heuristic State-Space Search", Heuristics Journal, Vol. 6, No. 4, Winter 1993, pp. 40-49.
[15] A.A. Toptsis, "Parallel Bidirectional Heuristic Search with Dynamic Process Re-Direction", Proc. 8-th International Parallel Processing Symposium, IPPS'94, IEEE Computer Society Press, April 1994, pp. 242-247.
[16] P.C. Nelson, and A.A. Toptsis, "Search Space Clustering in Parallel Bidirectional Heuristic Search", Proc. 4th UNB Artificial Intelligence Symposium, New Brunswick, Canada, September 1991, pp. 563-573.
[17] Applet for Voronoi diagrams, http://hirak99.googlepages.com/voronoi, (last accessed March 31, 2009).
[18] T. Kohonen, Self-Organizing Maps, Third extended edition. Springer, 2001.
[19] T. Kohonen, "Self-organized formation of topologically correct feature maps", *Biological Cybernetics*, Vol. 43, 1982, pp. 59-69.
[20] T. Kohonen and T. Honkela, "Kohonen Network", Scholarpedia, 2(1):1568, 2007.
[21] Georgy Voronoi, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques", Journal für die Reine und Angewandte Mathematik, 133, 1907, pp. 97-178.
[22] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, Spatial Tessellations - Concepts and Applications of Voronoi Diagrams, 2nd edition,. John Wiley, 2000.

[23] Franz Aurenhammer, “Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure”, ACM Computing Surveys, Vol. 23, No. 3, 1991, pp.345-405.
[24] M de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, Computational Geometry, third edition, Springer-Verlag, 2008.