UI-Route: An Ultra-Fast Incremental Maze Routing Algorithm

Tsung-Wei Huang*, Pei-Ci Wu†, and Martin D. F. Wong‡
*twh760812@gmail.com, †peiciwu@gmail.com, ‡mdfwong@illinois.edu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—Grid-based maze routing is a fundamental problem in electronic-design-automation (EDA) domain. A core primitive deals with a large query set about route connectivity subject to incremental changes on grid graph. Existing approaches pertain to batch processing, where each route query is independently and repeatedly solved by a routing procedure. Few researches so far discuss an efficient utilization of search knowledge in incremental fashion, which could dramatically speed up the search. Unfortunately, existing algorithms nearly rely on irregular and highly divergent search space, imposing acceleration challenges on refitting them to incremental version. Consequently, in this paper we present UI-Route, an ultra-fast incremental maze routing algorithm in grid environment. UI-Route is unique in breaking route equivalence, proving that a huge amount of equivalent search efforts can be optimally and incrementally eliminated. Equivalence breaking enables regularized search space, delivering well-tabulated search knowledge through the incremental processing. Moreover UI-Route is largely orthogonal to many applications built upon maze routing and therefore can seamlessly substitute for speedup. Experimental results on a set of modern circuit benchmarks demonstrate that UI-Route achieves prominent speedup over existing algorithms.**

I. Introduction

Grid-based maze routing is a fundamental problem in electronic-design-automation (EDA) domain [2], [8], [15]. Very common are applications dealing with signal connectivity between electronic components in order to design integrated circuits (ICs) or printed circuit boards (PCBs). A coding primitive typically involves a critical amount of subroutine calls of either route queries or incremental updates onto grid graph. A high-quality maze router is definitely positive to improve runtime bottleneck and tool scalability, especially for modern circuit designs which are far more dense and complex than last decades. Therefore, the goal in this paper is to revisit maze routing by a new incremental algorithm we have developed as well as a comparative investigation into existing algorithms under modern circuit benchmarks [11], [14].

Maze routing has received extensive research interests over the last decades [1], [3], [8], [13], [15]. Prior works focus on single routing instance, being either complete and optimal variants of breadth-first propagation (BFP) which often incur larger search space [4], [6], [9], or families of depth-first line search (DFL) which instead trade completeness or optimality for speedup [7], [10], [12]. Despite satisfactory performance for single routing instance, very few researches so far discuss an incremental processing in an efficient and compact manner as required in real applications. Existing approaches mostly rely on batch processing, in which each problem instance is fed as an independent input for the routing procedure. Unfortunately, algorithmic families like BFP and DFL have irregular and highly divergent search space, which imposes acceleration intractabilities on refitting them to incremental version. As a result, blind accumulation of duplicate search efforts is inevitable.

In this paper we present UI-Route, a new incremental maze routing algorithm motivated by explicit breaking of route equivalence on grid graph. Preserving the completeness and optimality, UI-Route adopts equivalence breaking strategies to regularize the search space. Regularized search space allows search knowledge to be well-tabulated, offering not only fast reflection to environmental changes but also huge savings from overlapped search efforts. A comparative example in Figure 1 demonstrates 2804$\times$ and 331$\times$ reductions on search efforts in terms of node expansions comparing UI-Route with two popular algorithms – Lee's algorithm and generic A* search [6], [9]. We highlight three key features of UI-Route: 1) it is simple and optimally efficient; 2) it implements incremental table lookup scheme to avoid duplicate search efforts, thereby contributing to significant speedup; 3) it is highly orthogonal to many applications built upon maze routing and thus can substitute for solid speedup.

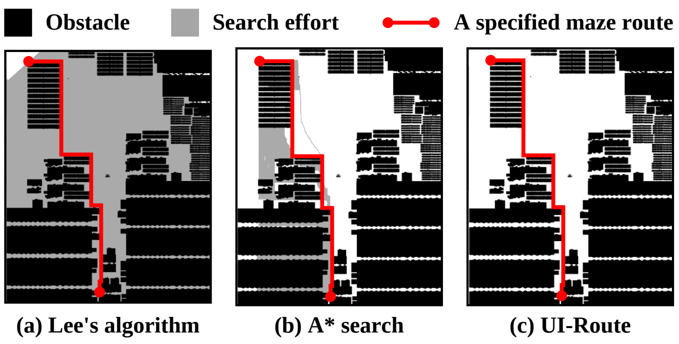


Figure 1. Comparison of search efforts on a circuit map. (a) Lee's algorithm requires 1323321 node expansions. (b) Regular A* search requires 209969 node expansions. (c) UI-Route requires only 635 node expansions.

Our contributions are summarized as follows: Firstly, we explicitly identify the route equivalence on grid graph and propose guidelines for equivalence breaking. By approaching route equivalence, we derive a compact and efficient incremental processing which speeds up not only the search itself but the entire class in conjunction with maze routing. Secondly, we prove the completeness and optimality of UI-Route, standing out above many literatures that usually trade either completeness or optimality for speedup. Thirdly, we run experiments on a set of industrial circuits released from recent ISPD contests [11], [14] and undertake an empirical analysis comparing UI-Route with existing maze routing algorithms. Comparatively, UI-Route demonstrates prominent speedup over existing algorithms. The experimental result can be an indicator assisting researchers in optimizing runtime bottleneck without loss of optimality.

SLIP '14, June 01 - 02 2014, San Francisco, CA, USA
Copyright 2014 ACM 978–1–4503–3053–4/14/06 $15.00.
http://dx.doi.org/10.1145/2633948.2633950

II. Incremental Maze Routing Problem

Consider a two-dimensional (2D) grid graph $G = \{V, E\}$, where V is the node set for grid entries and E is the edge set for neighboring connections. Non-traversable nodes such as on-grid obstacles and obstructions are also considered. In order to comply with most EDA applications, Manhattan routing direction (i.e., vertical/horizontal) is preferred. A maze route $R_{s\to t} = \langle S, v_i, \cdots, T\rangle$ is an ordered walk starting at source node S and ending at target node T. The route length is the number of edges connecting intermediate nodes along the route. A non-traversable node or a non-reachable path has infinity value. The problem input is a sequence consisting of three kinds of requests: 1) *Query*(S, T) asks the shortest route from the source node S to the target node T; 2) *Block*(g) blocks a traversable grid node g; 3) *Unblock*(g) unblocks a non-traversable grid node g. The problem formulation we defined in this paper is in fact online, i.e., prohibiting input disclosure for possible preprocessing, in order to reach more generality. An example of the incremental maze routing problem is given in Figure 2.

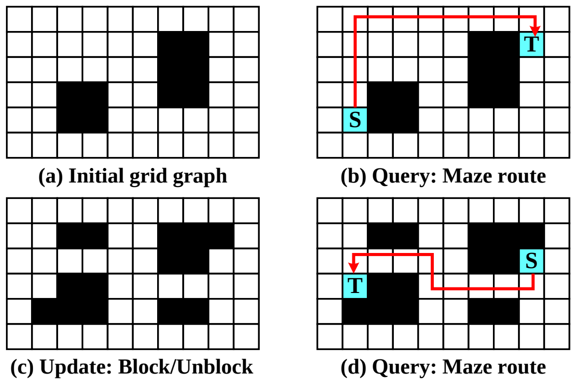


Figure 2. Incremental maze routing problem. (a) An initial grid graph. (b) A query about the maze route. (c) Graph update via blocking and unblocking. (d) A query about the maze route following the graph update.

We briefly highlight the necessities and difficulties of incremental maze routing problem.

1) **Practicality.** Most real applications require incremental processing, in which case having only one routing instance to be solved is less likely to occur but rather a large data subject to incremental changes on grid graph is designated.
2) **Knowledge extraction.** To achieve efficient incremental processing, consistent and regular information must be extracted from the search. We must be mindful that, as program progresses, such information might become more or less prolific.
3) **Portability.** To increase software portability, we are interested in the ease of implementation which imposes the least restriction on memory and computational requirements.

The goal of this paper is to represent the incremental processing by a data structure that offers knowledge about optimal search space and quick reflection to graph updates.

III. UI-Route

In this section we begin to introduce UI-Route. The key of UI-Route is to regularize the optimal search space by approaching route equivalence. We first discuss the idea of generic A* search and the explicit definition of route equivalence. Then we describe the technical implementation of UI-Route in bottom-up fashion.

A. Generic A* search

A* search is a general graph search algorithm that finds a least-cost route from a given source node to the target node [6]. A* search is featured by its scoring function, $f(x) = g(x) + h(x)$, applied for evaluating the cost of a current visiting node x. Note $g(x)$ is the cost from the source node to the visiting node x, and $h(x)$ is the "estimated" (or predicted) cost from the visiting node x to the target node. This concept is abstracted in Figure 3. Every time A* search picks up a node with the the lowest score (lowest $f(x)$) for expansion, i.e., exploring its neighbors for propagation. As a result, A* search is sometimes called the best-first search, because each expansion is prioritized on the route that is most likely to lead toward the target node. Generally speaking, Dijkstra's shortest path algorithm is a special case of A* search in which $h(x)$ is always equal to zero.

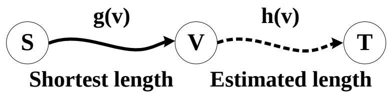


Figure 3. Concept of A* search.

A* search has twofold advantages. Firstly, since A* search adopts BFP yet framing it in a smarter way, it guarantees to find a route between any give source-target pair if one exists. Secondly, if $h(x)$ is admissible, meaning that it never overestimates the actual cost from the current node to the target node, then A* search is optimal. Therefore, for the Manhattan routing (i.e., only horizontal and vertical connections allowed), $h(x)$ is usually set as the Manhattan distance from any visiting node to the target, because it is the smallest possible distance between any two points in the Manhattan space.

B. Equivalent Routes and Equivalence Breaking

UI-Route is cored on approaching equivalent routes, selectively expanding only certain nodes that break route equivalence. We refer to these nodes as "*breaking nodes*," terming their nature of breaking route equivalence. We define the concepts of equivalent routes and breaking nodes in the following.

Definition 1: Given two distinct grid nodes g_1 and g_2, two routes r_1 and r_2 from g_1 to g_2 are equivalent, if they have identical lengths.

Definition 2: Given two distinct grid nodes g_1 and g_2, we define g_2 as a breaking node from g_1, if g_2 has at least one neighbor n such that g_2 is a must intermediate on the optimal route from g_1 to n.

Figure 4 illustrates our definitions for equivalent routes and breaking nodes. In (a), all specified maze routes have identical lengths and hence they are equivalent. In (b), the grid node marked by "B" is one possible breaking node from the leftmost turning node. This is because it has one neighbor, that is the target node below, whose optimal route heading from the leftmost turning node must include "B" as an intermediate. UI-Route is motivated by the two important concepts, searching over only a small set of breaking nodes to reach an optimal route.

C. Dimensional Extraction of Equivalence Breaking

Aforementioned concepts of equivalent routes and breaking nodes are in fact offline definitions. They require knowledge of node pairs and routes passing through to be given. Nonetheless, it is hard for normal search routine to directly identify route equivalence, since each expansion is reached by its neighbors piece-by-piece without any advance knowledge. Therefore, a rephrased definition of equivalence breaking is required.

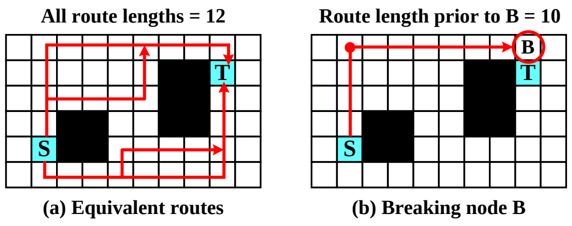


Figure 4. Illustration of definition 1 and definition 2. (a) Four equivalent routes. (b) A breaking node from the leftmost turning node (red solid circle).

Definition 3 - Horizontal breaking: Given a node g and its present search direction d in horizontal, the node g_H is a breaking node from g if node g_H is reachable from g by heading along direction d and is the nearest such node to satisfy one of the following conditions:

1) **Condition 1.** g_H is blocked or out of boundary.
2) **Condition 2.** g_H is traversable and at least one node g'_H vertically adjacent to the predecessor of g_H (i.e., one step prior to g_H) is blocked, while the node g''_H horizontally adjacent to g'_H in the direction d is traversable.

Condition 1 exerts a sanity check in which no equivalence information can be further extracted. Condition 2 essentially supports definition 2 for that g_H is a must intermediate on the shortest route heading from g to g''_H. In this regard, we are able to claim lemma 1, which can be illustrated in Figure 5.

Lemma 1: There are totally six cases of horizontal breaking nodes on any grid graph.

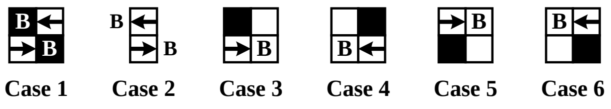
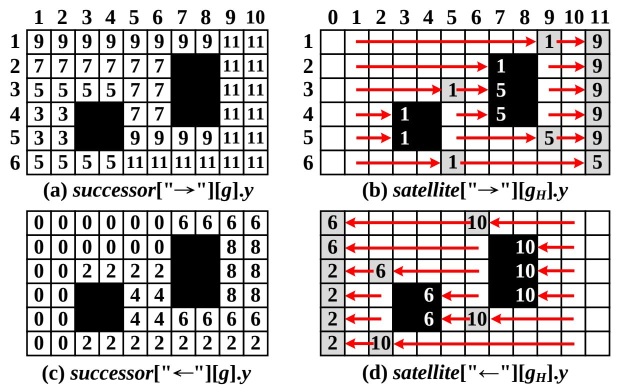


Figure 5. Six cases of horizontal breaking nodes (marked by "B"), where cases 1-2 and cases 3-6 illustrate the condition 1 and condition 2, respectively.

Using definition 3 as primitives, we are able to tabulate the horizontal breaking for quick information lookup and update. In particular, *successor* is a pointer table referring each node to its successive horizontal breaking node, while *satellite* is a sentinel table for each breaking node to keep track of its furthest predecessor for such breaking property to hold. An illustration is given in Figure 6, with numbers in tables indicating the y coordinate values accordingly. The numbers around the boundary denote the policy of our coordinate system (i.e., x and y correspond to vertical and horizontal coordinates in respective). In the earliest beginning, entries of *successor* and *satellite* tables are respectively initialized as NIL and grid nodes of respective entries.

Algorithm 1 presents the table lookup for horizontal breaking nodes. Given a node n and its one-step predecessor p along horizontal direction d, line 1 attempts to test whether a successive breaking node from p has been recorded ever. If yes, line 2:7 further examines the satellite field to verify the validity as it could be altered by graph updates. Otherwise, lemma 1 is recursively applied to renew the table (line 9 for cases 1–2 and line 10:15 for cases 3–6). On the basis of horizontal breaking, we define vertical breaking to fulfill the concept of equivalence breaking.

Definition 4 - Vertical breaking: A node g_V is vertical breaking from node g along vertical direction d', if g_V is the nearest node



	1	2	3	4	5	6	7	8	9	10
1	9	9	9	9	9	9	9	9	11	11
2	7	7	7	7	7	7	■	■	11	11
3	5	5	5	5	7	7	■	■	11	11
4	3	3	■	■	7	7	■	■	11	11
5	3	3	■	■	9	9	9	9	11	11
6	5	5	5	5	11	11	11	11	11	11

(a) *successor*["→"][*g*].*y*

0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6
1 9
1 9
1 5 9
1 5 9
1 5 9
1 5
(b) satellite["→"][g_H].y

0	0	0	0	0	0	6	6	6	6
0	0	0	0	0	0	■	■	8	8
0	0	2	2	2	2	■	■	8	8
0	0	■	■	4	4	■	■	8	8
0	0	■	■	4	4	6	6	6	6
0	0	2	2	2	2	2	2	2	2

(c) *successor*["←"][*g*].*y*

6 10
6 10
2 6 10
2 6 10
2 6 10
2 10
(d) satellite["←"][g_H].y

Figure 6. UI-Route tabulates the horizontal breaking information via two tables *successor* and *satellite*. Table *successor* records successive breaking nodes while table *satellite* monitors the corresponding validity. (a)-(b) Right direction. (c)-(d) Left direction.

from g such that there exists a horizontal breaking node by condition 2 from g_V in either horizontal direction d (i.e., $d' \perp d$).

Algorithm 1: TableLookup (p, n, d)

```
Input: nodes $p$ and $n$, and horizontal direction $d$
Output: the breaking node $g_H$ from node $p$ along direction $d$
1  if $successor[d][p] \neq NIL$ then
2      if $d =$ " $\rightarrow$ " and $satellite[d][successor[d][p]].y \leq p.y$ then
3          return $successor[d][p]$;
4      end
5      if $d =$ " $\leftarrow$ " and $satellite[d][successor[d][p]].y \geq p.y$ then
6          return $successor[d][p]$;
7      end
8  end
9  $successor[d][p] \leftarrow n$;
10 if $n.traversable$ then
11     if $n$ is a horizontal breaking node from $p$ along $d$ then
12         return $n$;
13     end
14     $successor[d][p] \leftarrow$ TableLookup($n$, $n.neighbor(d)$, $d$) ;
15 end
16 return $successor[d][p]$;
```

D. Incremental Processing for Graph Update

We begin the incremental processing by presenting the reflection to graph update. Without loss of generality, we deal with a single node as multiple updates can be decomposed into independent unit changes. We consider two kinds of updates, *Block* and *Unblock*, whereby a node is requested to be blocked or unblocked. As both operations are likely to affect the underlying structure of route equivalence, the goal is to provide quick reflection to this change with minimal tabular efforts imposed. We first introduce a subroutine, *shrink*, as in Algorithm 2, in which a non-preemptive max/min operation is applied to shrink the satellite field of a given horizontal breaking node g down to y with respect to the direction d.

Blocking and unblocking operations alter the underlying structure of horizontal breaking by three possible outcomes as sketched in Figure 7. Each outcome can be identified by referring to the difference between prior satellite field and present graph as described in Algorithms 3–4. Observing table *satellite* is one-to-one monitoring, we avoid renewing table *successor* as this manner incurs more tabular efforts, but rather shrink the satellite field of certain entries. In

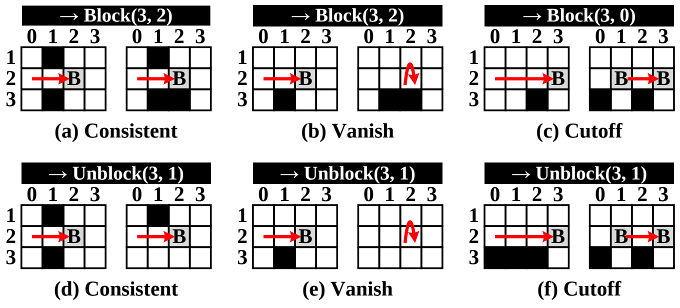


Figure 7. Three possible outcomes of horizontal breaking after unit graph update. (a)-(c) Case of node blocking. (d)-(f) Case of node unblocking.

Algorithm 2: shrink(g, d, y)

```
Input: a grid node $g$, direction $d$, and y coordinate value
1 if $g \neq NIL$ and $d =$ "$\rightarrow$" then
2     $satellite[d][g].y \leftarrow$ min($g.y$, max($satellite[d][g].y$, $y$));
3 end
4 if $g \neq NIL$ and $d =$ "$\leftarrow$" then
5     $satellite[d][g].y \leftarrow$ max($g.y$, min($satellite[d][g].y$, $y$));
6 end
```

Algorithm 5: Block(g)

```
Input: a traversable node $g$
1  Block the node $g$;
2  for $d \in \{$"$\rightarrow$", "$\leftarrow$"$\}$ do
3      $satellite[d][g] \leftarrow g$;
4      shrink($successor[d][g]$, $d$, $g.neighbor(d).y$);
5      $successor[d][g] \leftarrow$ NIL;
6  end
7  for $d \in \{$"$\searrow$", "$\swarrow$", "$\nearrow$", "$\nwarrow$"$\}$ do
8      if vanish(g.neighbor($d_x$), g.neighbor(d), reverse($d_y$)) then
9          shrink($g.neighbor(d_x)$, $reverse(d_y)$, $g.y$);
10     end
11     if cutoff(g.neighbor(d), g.neighbor($d_x$), $d_y$) then
12         shrink($successor[d_y][g.neighbor(d_x)]$, $d_y$, $g.neighbor(d).y$);
13     end
14 end
```

Algorithm 6: Unblock(g)

```
Input: a non-traversable node $g$
1  Unblock the node $g$;
2  for $d \in \{$"$\rightarrow$", "$\leftarrow$"$\}$ do
3      $satellite[d][g] \leftarrow g.y$;
4      $successor[d][g] \leftarrow$ NIL;
5  end
6  for $d \in \{$"$\searrow$", "$\swarrow$", "$\nearrow$", "$\nwarrow$"$\}$ do
7      if vanish(g.neighbor(d), g.neighbor($d_x$), $d_y$) then
8          shrink($g.neighbor(d)$, $d_y$, $g.neighbor(d).y$);
9      end
10     if cutoff(g.neighbor($d_x$), g.neighbor(d), reverse($d_y$) then
11         shrink($successor$[reverse($d_y$)][$g.neighbor(d)$], reverse($d_y$), $g.y$);
12     end
13 end
```

particular, Algorithms 5–6 carry out *constant time* manipulation. For convenience, $d_x(d_y)$ denotes unit $x(y)$ component of a diagonal direction d and *reverse*(d) denotes the mirror of a given Manhattan direction d. Algorithm 5 first blocks the satellite field of the blocked node (line 3) and any possible horizontal breaking passing through (line 4). The successor of this blocked node is no longer valid as well (line 5). Then Algorithms 3–4 are in turn applied to examine any change along with shrinkage performed if necessary (line 7:14). Likewise, Algorithm 6 first resets the satellite and successor fields of the unblocked node (line 2:5). The structural reflection to node unblocking is similar to the one in Algorithm 5, except for an opposite order of examination (line 6:13).

E. Route Query

Using Algorithms 1–6 as infrastructure, we develop the solution to *route query* – finding the shortest maze route from a given source node to a target node. Like most maze routing algorithms, the route distance and trace information are recorded in tables *dis* and *pie*, respectively. The key to establish a source-target connection is augmenting another tabular field for the target node. To achieve this goal, we introduce the following definition.

Definition 5 - Target breaking: The target node T belongs to both horizontal breaking and vertical breaking.

Target breaking is in fact a volatile definition. Since different route queries can have distinct target nodes, dedicated tabular field is required for each unique target node. This scenario is realized by Algorithm 7. Initiating from the target node T, the horizontal breaking successor of each node all the way to the leftmost and rightmost traversable nodes are redirected toward the target node, for purpose of creating target breaking (lines 6 and 14). Two backup storages (lines 5 and 13) are required for restoration when target breaking is being removed (lines 8 and 16). Had tabular field of target breaking been created, Algorithm 8 is applied to identify horizontal breaking nodes for successive search expansions. Rather than starting all over again, we adopt incremental search by calling Algorithm 1 to avoid duplicate search efforts on finding overlapped horizontal breaking nodes among different route queries (line 1). This step plays a pivotal role in speeding up the search, especially for frequent routes passing similar or identical regions. Recalling graph updates might result in incomplete satellite field, we need to renew such coverage so that every horizontal breaking node explored from the most recent search can be included (line 2:7). Finally, the horizontal breaking node undertakes the relaxation procedure and is returned if the corresponding distance label can be improved (line 8:14).

Algorithm 3: vanish(v, u, d)

```
Input: grid nodes $v$ and $u$, and direction $d$
Output: true if $v$ is vanished from $u$ along $d$ or false otherwise
1 if $satellite[d][v] = v$ then
2     return false;
3 end
4 if $v$ is a horizontal breaking node from $u$ along $d$ then
5     return false;
6 end
7 return true;
```

Algorithm 4: cutoff(v, u, d)

```
Input: grid nodes $v$ and $u$, and direction $d$
Output: true if $v$ cuts off the successor of $u$ along $d$ or false otherwise
1 if $satellite[d][v] \neq v$ then
2     return false;
3 end
4 if $v$ is not a horizontal breaking node from $u$ along $d$ then
5     return false;
6 end
7 return true;
```

Algorithm 7: setTargetBreaking(T, f)

```
   Input: the target node $T$ and a command flag $f$
1  for $d \in \{\leftarrow, \rightarrow\}$ do
2      $g \leftarrow T$;
3      while $g.traversable$ do
4          if $f$ = "CREATE" then
5              $backup$["successor"][$d$][$g$] $\leftarrow$ $successor$[$d$][$g$];
6              $successor$[$d$][$g$] $\leftarrow T$;
7          else
8              $successor$[$d$][$g$] $\leftarrow$ $backup$["successor"][$d$][$g$];
9          end
10         $g \leftarrow g.neighbor$(reverse($d$));
11     end
12     if $f$ = "CREATE" then
13         $backup$["satellite"][$d$][$T$] $\leftarrow$ $satellite$[$d$][$T$];
14         $satellite$[$d$][$T$] $\leftarrow$ $g.neighbor$($d$);
15     else
16         $satellite$[$d$][$T$] $\leftarrow$ $backup$["satellite"][$d$][$T$];
17     end
18 end
```

Algorithm 8: GetBreakingNode (p, n, d)

```
   Input: nodes $p$ and $n$, horizontal direction $d$
   Output: horizontal breaking node from $p$ along $d$
1  $n \leftarrow$ TableLookup($p$, $n$, $d$) ;
2  if $d =$ " $\rightarrow$ " and $p.y < satellite[d][n].y$ then
3      $satellite[d][n] \leftarrow p$;
4  end
5  if $d =$ " $\leftarrow$ " and $p.y > satellite[d][n].y$ then
6      $satellite[d][n] \leftarrow p$;
7  end
8  $\Delta y \leftarrow |n.y - p.y|$
9  if !$n.traversable$ or $dis[p] + \Delta y \geq dis[n]$ then
10     return $\phi$;
11 end
12 $dis[n] \leftarrow dis[p] + \Delta y$ ;
13 $pie[n] \leftarrow d$ ;
14 return $n$;
```

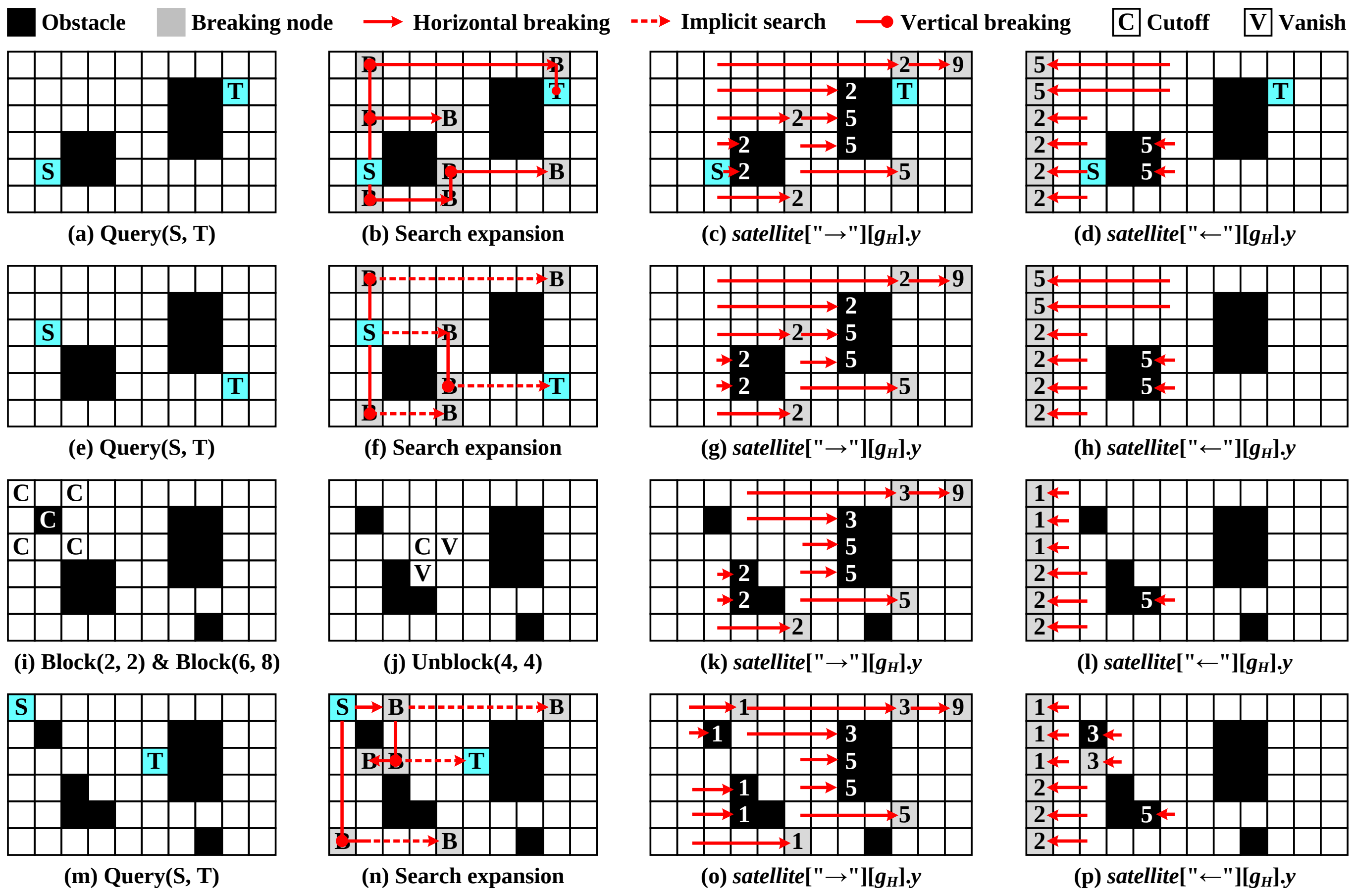
Algorithm 9: Query(S, T)

```
   Input: source node $S$ and target node $T$
   Output: shortest route $R_{s\rightarrow t}$ from source node $S$ to target node $T$
1  setTargetBreaking($T$, "CREATE");
2  Initialize $dis \leftarrow \infty$, $\pi \leftarrow$ NIL, $R_{s\rightarrow t} \leftarrow \phi$;
3  Priority queue $Q$ keyed on min$\{g \in Q | dis[g]$ + Heuristic$(g, T)\}$;
4  $dis[S] \leftarrow 0$;
5  $Q$.insert($S$);
6  while $Q \neq \phi$ do
7      $top \leftarrow Q$.extract();
8      if $top$ = $T$ then
9          $R_{s\rightarrow t} \leftarrow$ backtrack route from $T$ to $S$;
10         break;
11     end
12     for $d \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ do
13         $n \leftarrow top.neighbor(d)$;
14         if $d =$ " $\rightarrow$ " $or$ " $\leftarrow$ " then
15             $\Psi \leftarrow$ GetBreakingNode($top$, $n$, $d$);
16         else
17             $p \leftarrow top$;
18             while $n.traversable$ and $dis[p] + 1 < dis[n]$ do
19                 $dis[n] \leftarrow dis[p]$ + 1;
20                 $\pi[n] \leftarrow d$;
21                 $r \leftarrow n.neighbor$(" $\rightarrow$ ");
22                 $\Psi \leftarrow \Psi \cup$ GetBreakingNode($n$, $r$, " $\rightarrow$ ");
23                 $l \leftarrow n.neighbor$(" $\leftarrow$ ");
24                 $\Psi \leftarrow \Psi \cup$ GetBreakingNode($n$, $l$, " $\leftarrow$ ");
25                 if $\Psi \neq \phi$ or $n = T$ then
26                     $\Psi = \Psi \cup n$;
27                     break;
28                 end
29                 $p \leftarrow n$;
30                 $n \leftarrow n.neighbor(d)$;
31             end
32         end
33         for $g \in \Psi$ do
34             $Q$.enque($g$);
35         end
36     end
37 end
38 setTargetBreaking($T$, "REMOVE");
39 return $R_{s\rightarrow t}$;
```

The pesudocode of our search algorithm is presented in Algorithm 9, which is structured by generic A* framework. Prior to the search, Algorithm 7 is called to create dedicated tabular field for the target breaking (line 1). The priority queue is keyed on A* score and Manhattan distance is our default heuristic estimation (line 3). The major search loop (line 6:37) iteratively looks for a node with lowest A* score from the priority queue and extracts it out for expansion (line 7). Iteration stops when the target node is ever extracted out, in which the optimal maze route has been found (line 8:11), or no more queuing operations remained in which no path exists. During each expansion, we attempt to identify breaking nodes, instead of blindly picking up neighboring nodes, for successive expansions (line 14:32). If the present search direction is horizontal, we directly call Algorithm 8 to attain horizontal breaking nodes (line 14:16). If the present search direction is vertical, we incrementally apply Algorithm 8 for each vertical step until the first successful return of vertical breaking by definition 4, or no further step can be proceeded (line 17:31). In the end of expansion, we perform queuing operations only on breaking nodes to minimize the queuing efforts (line 33:35). Since the target breaking is only active for this search, the corresponding tabular field needs to be restored to previous values prior to the return of route information (line 38:39).

F. Exemplification

The algorithmic procedure of UI-Route can be visualized in Figure 8. For concise illustration, we do not list the *successor* table as it can be inferred by the *satellite* correspondences. The coordinate system used in this exemplification is the same as Figure 6. We use solid lines to suggest the explicit search expansion in which all nodes along the line account for explorations of horizontal breaking nodes, and draw dashed lines for implicit search where the horizontal breaking successor is identified via table lookup. In the initial stage, all entries of satellite table are initiated as the corresponding nodes. We can see in (a)–(d) UI-Route begins with a fresh search, exploring all required breaking nodes for the search expansion, meanwhile storing the locations of horizontal breaking nodes into the table. The existence of vertical breaking node is referred to any traversable horizontal breaking node along either horizontal search direction. For example in (b), the node below the source node is a vertical breaking node from the source, since it has a horizontal breaking successor at (6, 5). The benefit of tabulating horizontal breaking is clearly exhibited when dealing with the second route query in (e). As shown in (f)–(h), the solution space to this query shares almost the same set of horizontal breaking nodes with the previous query, whose locations have been recorded and thus can be quickly identified by table lookup for the subsequent search. In other words, the search routine



Obstacle
Breaking node
Horizontal breaking
Implicit search
Vertical breaking
C Cutoff
V Vanish
(a) Query(S, T)
(b) Search expansion
(c) *satellite*["→"][g_H].*y*
(d) *satellite*["←"][g_H].*y*
(e) Query(S, T)
(f) Search expansion
(g) *satellite*["→"][g_H].*y*
(h) *satellite*["←"][g_H].*y*
(i) Block(2, 2) & Block(6, 8)
(j) Unblock(4, 4)
(k) *satellite*["→"][g_H].*y*
(l) *satellite*["←"][g_H].*y*
(m) Query(S, T)
(n) Search expansion
(o) *satellite*["→"][g_H].*y*
(p) *satellite*["←"][g_H].*y*

Figure 8. Exemplification of UI-Route. (a) The first route query. (b) UI-Route begins with a fresh search, exploring vertical breaking nodes and horizontal breaking nodes for search expansion. (c)–(d) During the search, the locations of horizontal breaking nodes explored for this search are stored in table. (e) The second route query. (f) The second search routine performs implicit search on horizontal breaking nodes via table lookup to avoid duplicate search efforts. (g)–(h) The table remains unchanged for no new horizontal breaking nodes introduced by this search. (i) Cutoff cases by blocking. (j) Vanish and cutoff cases by unblocking. (k)–(l) Structural changes of underlying horizontal breaking are reflected by shrinking the satellite field. (m) The third route query. (n) The search routine explores a few extra horizontal breaking nodes to obtain the shortest route. (o)–(p) Besides the prior tabular field, several entries are incrementally renewed to cover newly found horizontal breaking nodes.

can incrementally jump over horizontal breaking nodes instead of explicitly searching each of them all over again. This scenario is the key for UI-Route to speed up the search, especially when similar or identical routes passing through common region. An example of graph update is shown in (i)–(l). We can see in (i) the blocked node at (2, 2) introduces five cutoff cases on present grid graph, while the blocked node at (6, 8) has no impact on the current satellite field as a result of consistent case. On the other hand, the unblocked node at (4, 4) vanishes, in addition to itself, the horizontal breaking node passing through its diagonal corner at (3, 5). After applying the shrink procedure, the satellite field of each affected horizontal breaking node is reflected as (k) and (l). Notice that both successor and satellite fields could be incomplete since UI-Route fills in the two tables in incremental fashion. Only fields necessitated for solving a given route query are involved in tabular manipulations. Finally, in dealing with the third route query, a few horizontal breaking nodes are additionally explored by the search routine along with an incremental update of the corresponding tabular fields as shown in (m)–(p).

IV. Completeness and Optimality of UI-Route

To prove the completeness and optimality of UI-Route, we claim two key features: 1) the optimal route appears in the search space of UI-Route; 2) once the target is expanded (i.e., extracted out of the priority queue), UI-Route has found an optimal route. For the sake of clarity, we decompose Theorem 1 into two sections by separating completeness and optimality.

Theorem 1-1: UI-Route is complete.

Proof: The input grid graph we considered in this paper is indeed a locally finite graph, which means there is only a finite number of paths with finite length values thorough the entire search. Specifically, the successor table stores the information that can be exactly viewed as a segment set consisting of horizontal breaking nodes. By travelling over these horizontal breaking nodes, we are able to virtually visit all nodes during the search routine. This fact, as well as definition 5, means that the target node must ever participated in queuing operations during the entire search if at least one source-target route exists. If not by assumption, there is at least one another path after the search eventually ends, which contradicts the fact that only a finite number of paths thorough the entire search. Therefore, UI-Route guarantees to return a route if one exists. ■

Theorem 1-2: UI-Route is optimal.

Proof: Proving the optimality of UI-Route is equivalent to showing that once a node is extracted out of the priority queue, UI-Route has found its optimal route. Since our heuristic is scored on Manhattan distance which is admissible for A* search, we shall focus on ordinary search procedure without heuristic score involved. We should be also aware that the distance between successive breaking nodes is always shortest due to the path monotonicity. Therefore we can narrow down the scope of proof on breaking nodes. Let v_{ext} be the node immediately extracted out of the priority queue and v_{pri} be the node ever extracted out prior to v_{ext}. The prove is to show $dis[v_{ext}]$ is optimal. Assuming the $dis[v_{ext}]$ is not optimal, then there exists another path passing through v_{pri} and at least one intermediate node v' ($v' \neq v_{pri}$) such that the total route cost to v_{ext} is less than $dis[v_{ext}]$. In other words, $dis[v']$ should be less than $dis[v_{ext}]$. By definitions 3–5, such an intermediate node must be a breaking node that remains in the priority queue. However, this contradicts the min property of priority queue since node v_{ext} had been extracted out prior to node v'. Therefore by contradiction $dis[v_{ext}]$ is optimal. As our heuristic score is admissible, framing the above proof by A* search has no impact on the optimality of UI-Route [6]. ■

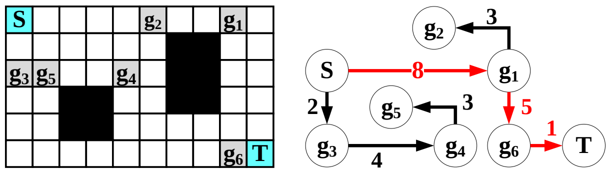


Figure 9. A partial solution trace demonstrates the search space of UI-Route is only optimal for source-target connection.

The proof is true on the basis of breaking nodes, rather than all traversable nodes as ordinary occasion. To be more precisely, UI-Route is only optimal for peer-to-peer shortest connection instead of all intermediates. An example of solution fragment is shown in Figure 9. In the normal case, the shortest route from source node S to node g_2 is 5, whereas in the search space of UI-Route it turns out to be 11. The reason is that g_2 is a horizontal breaking node from g_1 and g_1 is in turn a horizontal breaking node from S. Another identical case is the search passing from g_4 to g_5. It is fair to claim that UI-Route casts off the conventional shortest path structure and gains speedup by searching over a relatively small set of breaking nodes that are effective enough to construct the optimal route. From this point of view, the speedup is obvious.

V. Experimental Evaluation

We implement UI-Route in C++ language on a 2.67GHz 64-bit Linux machine with 8GB memory. Experiments are undertook on industrial benchmarks released from recent ISPD contests [11], [14]. Each circuit benchmark is linearly scaled into a two-dimensional grid graph with maximum dimension equal to 2048. For each circuit benchmark, an input sequence consisting of block, unblock, and query operations is extracted from the following procedure: We iteratively and randomly generate a route query and apply Lee's algorithm to solve it. If any exists, the route returned is blocked as physical obstruction. Otherwise, we loop backward and unblock the previous route until the query can be resolved. Iteration ceases when a total of 1000 route queries have been solved. The experiment is meaningful since it simulates an incremental procedure that is widely employed in many EDA applications.

A. Baseline Algorithms

In order to perform a controlled analysis, we limit the score of comparisons to batch processing using maze routing algorithms from EDA domain. Considering the page restriction, we are unable to report all state-of-the-art but rather three representative – Lee's algorithm [9], A* search [6], and Soukup's algorithm [12], which turn out to be the most stable and efficient in respective BFP and DFL families according to our implementations [1], [4], [5], [6], [7], [9], [10], [12]. In fact, Soukup's algorithm is the fastest over all trials. The performance gap between optimal and suboptimal algorithms is clear enough by comparing with Soukup's algorithm. We quantify performance in terms of average and maximum speedup acquired to solve one query in relative to Lee's algorithm. The accumulated runtime has similar acceleration trend which approaches to the average speedup value when input query keeps feeding. Since UI-Route carries constant time overhead per unit graph update, the time is negligible. For Soukup's algorithm, we measure the suboptimality by relative error in percentage.

B. Performance Comparison

We begin by comparing UI-Route with A* search algorithm. As in Table I, the incremental processing of UI-Route exhibits convincing speedup across all benchmarks over existing algorithms. The largest difference is observed on superblue12 where UI-Route reaches the goal by 10.7× faster, while A* search only acquires 5.9× speedup. We conclude that even though A* search can generally reduce the search efforts, there remains a far performance gap between batch processing and incremental counterpart. As most BFP variants, A* search has worst case of virtually exploring all nodes in which situation, nevertheless, extra priority queue operations overwhelmed the algorithmic advantages (see Figure 10). From this point of view, equivalence breaking and tabular strategies prevent UI-Route from blind accumulation of duplicate search efforts but otherwise a small set of breaking nodes. As consequence, UI-Route achieves relatively stable and promising reduction on search efforts by at least two orders of magnitude, outperforming A* search across all circuit benchmarks.

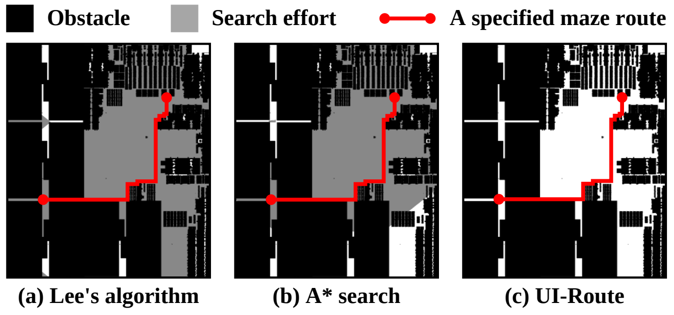


Figure 10. An example from superblue2 shows that the reduced search efforts by A* search are unable to amortize the time spent on queuing operations sufficiently, resulting in an inferior speedup by 0.36×.

Next, we compare UI-Route with Soukup's algorithm. Unlike A* search which is much slower than Soukup's algorithm, UI-Route demonstrates comparable performance margin or even faster in some cases such as adaptec3 and superblue15. Despite superior speedup, the greedy DFL process excludes Soukup's algorithm from optimal family. The maximum error could reach up to 254% in superblue10 which is not acceptable for many wirelength-driven applications [14]. On contrast, UI-Route proves that by equivalence breaking a huge

TABLE I
PERFORMANCE COMPARISON BETWEEN UI-ROUTE AND EXISTING MAZE-ROUTING ALGORITHMS.

Circuit	Grid Size	Space	A* search [6]				Soukup [12]						UI-Route			
			ASP	MSP	ARS	MRS	ASP	MSP	ARS	MRS	ARE	MRE	ASP	MSP	ARS	MRS
adaptec1	2048×2048	51%	3.9	17.4	25	196	7.2	30.1	32242	330047	3	74	6.8	25.4	50210	351188
adaptec2	2048×2048	36%	2.8	7.9	18.5	277.6	4.5	15.7	7405	133194	3	90	4.8	15.5	17105	241327
adaptec3	2048×2031	37%	2.9	7.2	15	76	4.8	14.8	964	145261	6	110	5.5	17.3	6090	59829
adaptec4	2048×2034	50%	3.9	10.7	20	122	6.7	28.1	1696	148285	6	101	7.1	25.8	10854	85404
bigblue1	2048×2048	72%	6.0	23.1	43	527	11.0	38.1	172538	532998	1	94	10.6	36.8	162449	1080055
bigblue2	2048×2040	59%	3.1	23.2	14	74.8	4.6	35.1	437	133599	9	187	5.8	28.3	1087	33682
bigblue3	2048×2036	32%	2.9	10.0	17.7	200	4.4	22.2	3800	85670	3	64	4.7	19.9	6216	312364
bigblue4	2048×2038	60%	4.5	14.5	26	100	6.7	30.3	424	43841	4	50	6.2	23.8	832	12129
superblue1	1502×2048	44%	3.5	15.6	23	396	5.9	23.9	20507	373452	2	163	5.7	23.2	15209	400133
superblue2	2048×1417	32%	2.3	10.8	14	775	3.6	20.0	8612	177841	3	154	3.8	14.9	8240	200991
superblue4	2048×1843	52%	4.0	13.8	23	465	7.5	21.1	38056	604577	2	150	6.9	24.4	31061	741609
superblue5	1885×2048	40%	2.4	13.8	17	239	3.6	17.8	15354	230696	3	235	3.8	16.2	4326	97962
superblue10	2048×1349	37%	3.2	9.6	19	280	5.5	20.1	6408	134373	4	254	4.8	15.0	4163	122562
superblue12	2048×1407	78%	5.9	21.7	36	369	11.5	56.8	121501	3677116	0	106	10.7	48.3	92905	1003173
superblue15	2048×1321	62%	3.6	13.4	18	234	6.6	26.9	7788	302598	10	199	7.2	30.4	9436	175248
superblue18	2048×1546	62%	5.4	17.3	31	367	10.8	41.9	55326	349027	2	68	9.7	40.6	39264	704330

ASP/MSP: Avg/Max speedup of CPU time (times). ARS/MRS: Avg/Max reduction on search efforts (times). ARE/MRE: Avg/Max relative error of suboptimality (%).

amount of search efforts can be incrementally skipped without loss of optimality.

VI. CONCLUSION

In this paper we have presented UI-Route to approach incremental maze routing problem. Unlike conventional batch processing, UI-Route performs incremental search via explicit equivalence breaking to avoid blind accumulation of duplicate search efforts. The proposed guidelines of equivalence breaking speed up not only the search itself but the entire domain in conjunction with incremental maze routing. UI-Route has several merits such as simplicity, coding ease, and most importantly the theoretically-proved completeness and optimality. Besides, UI-Route is highly orthogonal to many applications built upon maze routing and therefore can easily substitute for solid speedup. Experimental results on modern circuit benchmarks have demonstrated prominent speedup of UI-Route over existing algorithms. Finally, we briefly highlight several implementation details and trials that are worth delivering as follow:

- **Clever data structure matters.** We have implemented two different data structures, binary heap and bucket heap, for priority queue operations. The bucket heap supports relatively fast queuing operations than binary heap since most computations are done within local array access. We observe using bucket heap that UI-Route is able to improve speedup by about 17%. In other words, the time reported in this paper should be viewed as a generous upper bound on a custom implementation.
- **Hidden experiments.** We have implemented a vast of algorithms while hiding most of them from discussion for scope efficiency. These implementations include open entries such as JPS and Tree Cache released in 2012–2013 Grid-Based Path Planning Competition (GPPC) [1]. However, these entries require diagonal angles to be enabled and hence are less suitable for most applications in EDA domain.
- **Generalization to weighted grid graphs.** We have attempted to suit UI-Route with weighted grid graphs. We alter condition 2 by checking the cost value estimated so far to identify horizontal breaking nodes locally. The entire algorithm functions correctly while speedup is not remarkable. We reason that various cost distributions might force equivalent routes to exhibit wild swings in the near future, rather than simply a branch from present visiting node as uniform environment. In this case, a non-negligible amount of computation efforts are spent on identifying breaking nodes. A more sophisticated cutoff strategy needs to be invented in the future work.

REFERENCES

[1] GPPC 2013: Grid-Based Path Planning Competition, Moving AI Lab., http://movingai.com/GPPC/

[2] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, "Handbook of Algorithms for Physical Design Automation," CRC Press, 2009.

[3] T. Cazenave, "Optimizations of Data Structures, Heuristics and Algorithms for Path-finding on Maps," *IEEE symp. on CIG*, pp. 27–33, 2006.

[4] F. O. Hadlock, "A Shortest Path Algorithm for Grid graphs," *Networks*, vol. 7, pp. 323334, 1977.

[5] D. Harabor and A. Grastien, "Online Graph Pruning for Pathfinding on Grid Maps," *Prof. AAAI*, 2011.

[6] P. E. Hart, N. J. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on SSC*, vol. 4, pp. 100–107, 1968.

[7] D. Hightower, "A Solution to Line Routing Problems on the Continuous Plane," *Proc. ACM/IEEE DAC*, pp. 1–24, 1969.

[8] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, "VLSI Physical Design: From Graph Partitioning to Timing Closure," *Springer*, 2011.

[9] C. Y. Lee, "An Algorithm for Path Connection and its Application," *IEEE Trans. on Electronic Computers*, vol. 10, pp. 346–365, 1961.

[10] K. Mikami and K. Tabuchi, "A Computer Program for Optimal Routing of Printed Circuit Connectors," *Proc. Int. Federation for Information Processing*, pp. 1475–1478, 1968.

[11] G.-J. Nam, "ISPD 2006 Placement Contest: Benchmark Suite and Results," *Proc. ACM ISPD*, pp. 167, 2006.

[12] J. Soukup, "Fast Maze Router," *Proc. ACM/IEEE DAC*, pp. 100–102, 1978

[13] N. R. Sturtevant, "Benchmarks for Grid-Based Pathfinding," *IEEE Trans. on CIAIG*, vol. 4, pp. 144–148, 2012

[14] N. Viswanathan, C. J. Alpert, C. Sze, Z. Li, G.-J. Nam, and J. A. Roy, "The ISPD-2011 Routability-Driven Placement Contest and Benchmark Suite," *Proc. ACM ISPD*, pp. 141–146, 2011.

[15] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, "Electronic Design Automation: Synthesis, Verification, and Testing," Elsevier, 2009.