Effective Front-to-Front Heuristics in Bidirectional Search

Shivam Dharmeshkumar Shah
Bachelor of Science in Computer Science and Information Systems
Supervisor: Michael Barley and Pat Riddle
Department of Computer Science
University of Auckland

February 12, 2018

It is with absolute delight and the deepest gratitude that I receive this scholarship to assist me with this research. These 3 months of research have been amazing, I learned so much from my supervisors who have seen me scribbling and writing down the ideas of "where am I going wrong" to actually implementing and seeing them work. I have always believed in “A journey of a thousand begins with a single step” and this research was my first step. Finally, I would say I enjoyed working on something that really interests me and might potentially further my career in it.

1. Introduction

Bidirectional search is an algorithm that uses two searches occurring at the same time to reach a target goal. It generally appears to be efficient search algorithm as instead of searching deep through the graph, we conduct search from both the ends; forward from start and backward from goal and the reason gave it to be faster is because the area covered by two small searches from both ends is much smaller than area covered from one end. The major problem was the intersection of two searches when the graph is dense. To overcome that problem Bidirectional Heuristic front to front algorithm(BHFFA) which guaranteed that two frontiers will collide, however it turned out to be computationally taxing; both in time and space. Furthermore, this taxing computation makes our current approach of using front-front heuristics far less effective than both blind and front-end bidirectional search in terms of reducing problem-solving time. Subsequently, this research answers several important questions like how much can front-front reduce the size of the explored search space compared to front-end and blind search, also how expensive the naïve approaches are and lastly how effective is front-front heuristic search in compare to front-end heuristic.

2. Related Research

(An Optimality theorem for a bi-directional search algorithm) report talks about bi-directional heuristic search algorithm’s theorems, focus here is the heuristic we use i.e. if the heuristic used is ‘bad’ then it will expand at least the nodes that are expanded using ‘good’ heuristic. Then it tries the generalize and give an optimality theorem for bi-directional heuristic search by defining two theorems of which Theorem 1 states if the estimated distance from some node x to some node y is less than or equal to the minimum distance from x to y and all the edge weights are no less than some positive constant than the bi-directional heuristic search algorithm halts with a shortest path between start and goal (if it exists). This paper gave some insights on how ‘good’ or ‘bad’ heuristic function can impact the overall algorithms optimality.

The overall idea of using abstraction instead of any Heuristic function was my primary goal. (Searching Without a Heuristic: Efficient Use of Abstraction, 2010) paper helped me to understand how abstraction can be used to derive admissible heuristic values without using traditional heuristic functions. The abstraction of a search problem is a simplification of that problem such that the minimum cost between two nodes in the abstract space is no greater than the minimum cost between the corresponding nodes in the original problem. A pattern-database utilizes that idea, a pattern-database (PDB) is a look-up table that maps abstract problem states to admissible heuristic cost-to-go estimates for the original state problem. There are significant drawbacks to this approach, PDBs are generated by performing an exhaustive uniform-cost search in the abstract space, this takes hours to compute a precise abstraction. Also, it consumes a large amount of memory which can no longer be used for storing search nodes. An alternative and a better approach is to use hierarchical heuristic search which computes the database lazily i.e. the cost of the optimal solution in abstract space can be used in the lower level as a good estimate. Although, a naïve implementation of this is practically inefficient since many searches in the abstract space can be used again in the future. Hence to improve the efficiency, several caching techniques have been devised.

Furthermore, this (Searching Without a Heuristic: Efficient Use of Abstraction, 2010) paper introduces a new way of using hierarchical heuristic called Switchback that expands nodes at most once. It addresses the major node re-expansion. It does it by alternating forward and backward searches from one level to the next. The benefit of using it is that every single expansion at the abstract level will contain an optimal path from the expanded node to the abstraction of the goal node in the level below, and in future, if the value is required then one can simply retrieve it from a lookup table. This paper also proves the heuristic used by switchback is both consistent and admissible at every level of abstraction and they also conclude that switchback outperforms both HIDA* and HA* on four domains they used. This paper led me to use the idea of hierarchical heuristic for both front-end and front-front Bidirectional search which is further discussed in the report.

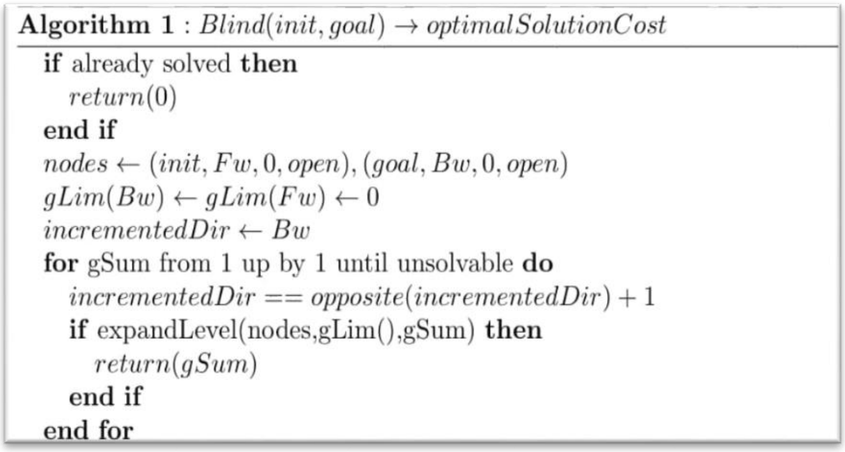
Another interesting (Faster Optimal and Suboptimal Hierarchical Search, 2011) paper talks about Hierarchical A*, which performs A* search at each level of abstraction. Conventional A* is run on the base level by removing a node with smallest f value, generating its children finding their heuristic values. To find a heuristic value of a node at base space, the node is abstracted at next level and A* is started from that node to abstract goal node, the terminated path length will be the heuristic value for the base node. This is all done recursively, the highest level will most trivial space. The problem here is the node re-expansion, to solve that it uses three caching techniques (This could potentially be the next thing I will be doing to improve my front-end and front-front algorithms that uses abstraction). Furthermore, it talks about short circuit (Something I am interested to test it in future).

3. Experiments

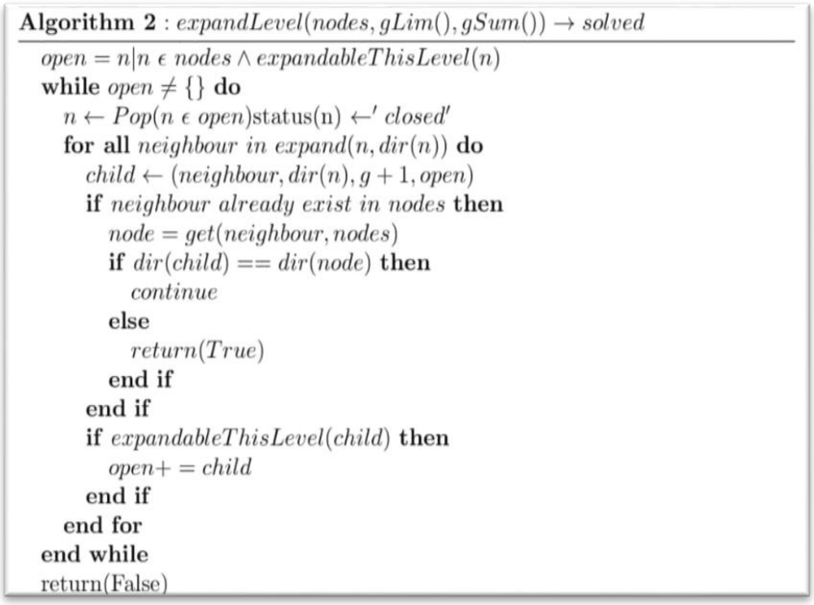
Approach 1: Blind Bidirectional Search

Pseudocode for Blind search is given in Figure 1. Algorithm 1 takes in the initial state, desired goal state and computes the optimal solution cost. We do so by alternating between forward and backward directions and incrementally increasing the level (gSum) by expanding all the nodes for that level until we can't find a solution. In Algorithm 2, we create a temporary data structure which holds all the nodes which can be expanded at that level and check to find a collision node (i.e. a node that exist in both forward and backward frontier).

The domain here is 8-Puzzle and we limit our problem set to 100 problems per solution path (with solution cost ≤ 21).



```
Algorithm 1 : $Blind(init, goal) \rightarrow optimalSolutionCost$
  if already solved then
    $return(0)$
  end if
  $nodes \leftarrow (init, Fw, 0, open), (goal, Bw, 0, open)$
  $gLim(Bw) \leftarrow gLim(Fw) \leftarrow 0$
  $incrementedDir \leftarrow Bw$
  for gSum from 1 up by 1 until unsolvable do
    $incrementedDir == opposite(incrementedDir) + 1$
    if expandLevel(nodes,gLim(),gSum) then
      $return(gSum)$
    end if
  end for
```



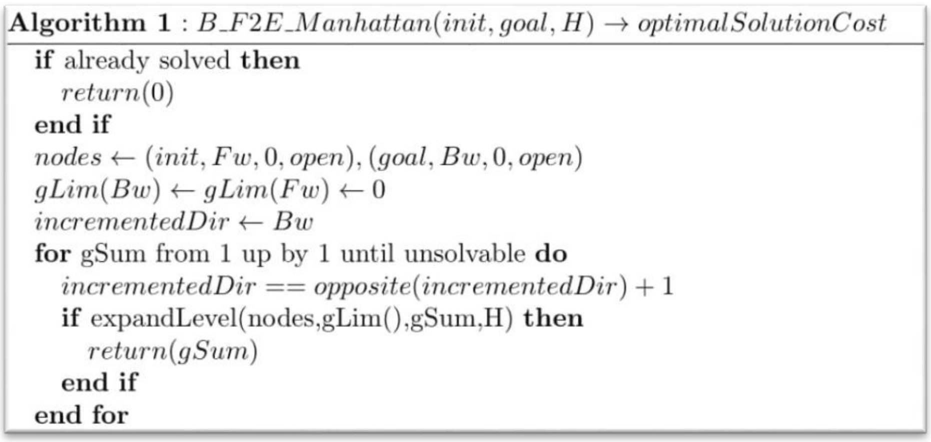
```
Algorithm 2 : $expandLevel(nodes, gLim(), gSum()) \rightarrow solved$
  $open = n|n \epsilon nodes \wedge expandableThisLevel(n)$
  while $open \neq \{\}$ do
    $n \leftarrow Pop(n \epsilon open)$status(n) $\leftarrow' closed'$
    for all $neighbour\ in\ expand(n, dir(n))$ do
      $child \leftarrow (neighbour, dir(n), g + 1, open)$
      if $neighbour\ already\ exist\ in\ nodes$ then
        $node = get(neighbour, nodes)$
        if $dir(child) == dir(node)$ then
          $continue$
        else
          $return(True)$
        end if
      end if
      if $expandableThisLevel(child)$ then
        $open+ = child$
      end if
    end for
  end while
  return(False)
```

Fig. 1. Pseudocode for Blind Bidirectional Search

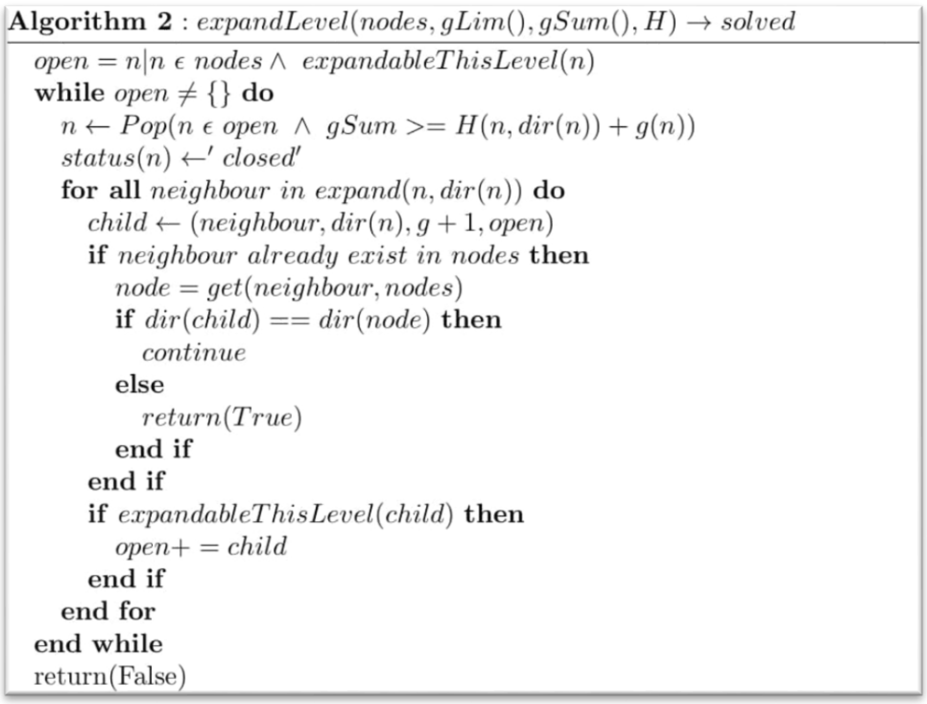
Approach 2: Bidirectional Front-End Search using Manhattan Distance

Pseudocode for Front-End using Manhattan Distance heuristic is given in Figure 2. The algorithm is identical to blind except we use H(s,t) (i.e. Heuristic function which gives an estimate of the minimum cost from node s to the goal t), to calculate Heuristic value we use admissible and consistent heuristic called Manhattan Distance. In Algorithm 2, we call our function H(n, dir(n)) which calculates the Manhattan distance from state n to goal state (if the direction of the node is 'forward', else if the direction is 'backward' we find the distance from state n to start state).

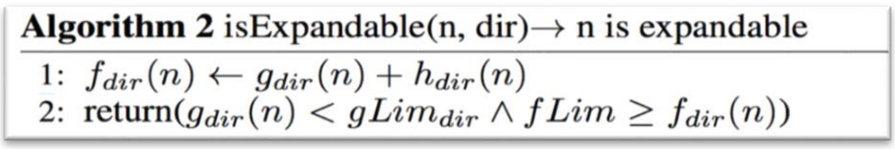
To check if a node is expandable at that level, we compute f(n) = g(n) + h(n) (where f(n) is the actual path cost, g(n) cost to reach that node and h(n) is an estimate to goal) and test to see whether gSum ≥ f(n) and g(n) < glim(n). Furthermore, domain for here is same as our approach 1.



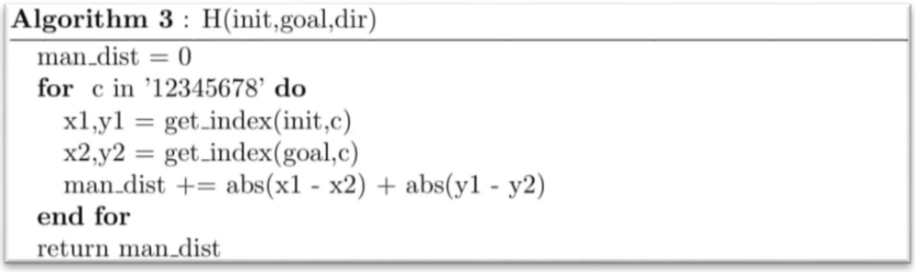
```
Algorithm 1 : $B\_F2E\_Manhattan(init, goal, H) \rightarrow optimalSolutionCost$
  if already solved then
    $return(0)$
  end if
  $nodes \leftarrow (init, Fw, 0, open), (goal, Bw, 0, open)$
  $gLim(Bw) \leftarrow gLim(Fw) \leftarrow 0$
  $incrementedDir \leftarrow Bw$
  for gSum from 1 up by 1 until unsolvable do
    $incrementedDir == opposite(incrementedDir) + 1$
    if expandLevel(nodes,gLim(),gSum,H) then
      $return(gSum)$
    end if
  end for
```



```
Algorithm 2 : $expandLevel(nodes, gLim(), gSum(), H) \rightarrow solved$
  $open = n | n \epsilon nodes \wedge expandableThisLevel(n)$
  while $open \neq \{\}$ do
    $n \leftarrow Pop(n \epsilon open \wedge gSum >= H(n, dir(n)) + g(n))$
    $status(n) \leftarrow' closed'$
    for all $neighbour\ in\ expand(n, dir(n))$ do
      $child \leftarrow (neighbour, dir(n), g + 1, open)$
      if $neighbour\ already\ exist\ in\ nodes$ then
        $node = get(neighbour, nodes)$
        if $dir(child) == dir(node)$ then
          $continue$
        else
          $return(True)$
        end if
      end if
      if $expandableThisLevel(child)$ then
        $open+ = child$
      end if
    end for
  end while
  return(False)
```



```
Algorithm 2 isExpandable(n, dir)→ n is expandable
1: $f_{dir}(n) \leftarrow g_{dir}(n) + h_{dir}(n)$
2: return($g_{dir}(n) < gLim_{dir} \wedge fLim \geq f_{dir}(n)$)
```

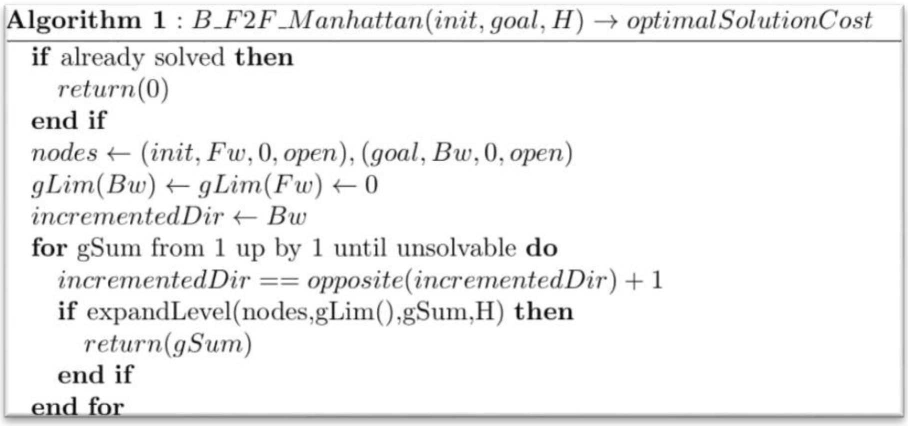


```
Algorithm 3 : H(init,goal,dir)
  man_dist = 0
  for c in '12345678' do
    x1,y1 = get_index(init,c)
    x2,y2 = get_index(goal,c)
    man_dist += abs(x1 - x2) + abs(y1 - y2)
  end for
  return man_dist
```

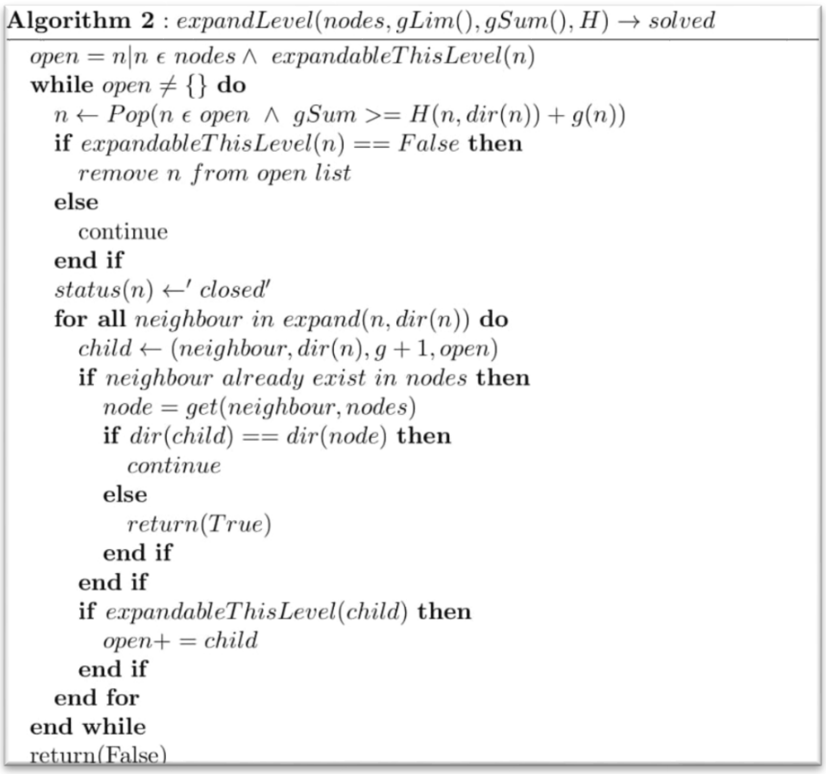
Fig. 2. Pseudocode for Bidirectional Front-End using Manhattan

Approach 3: Bidirectional Front-Front Search using Manhattan Distance

This approach is fairly similar to our approach 2, except now instead of calculating the heuristic from node n to the goal we calculate from node n to all the nodes that are open in opposite frontier. Furthermore, since the heuristic values computed are dynamic, we need to recheck if the node popped from the open list can be expanded as shown in Algorithm 2 of Figure 3. The domain we chose here is same as approach 1.

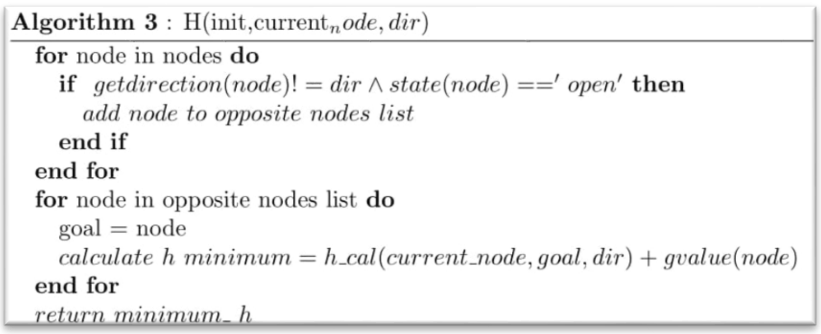


```
Algorithm 1 : $B\_F2F\_Manhattan(init, goal, H) \rightarrow optimalSolutionCost$
  if already solved then
    $return(0)$
  end if
  $nodes \leftarrow (init, Fw, 0, open), (goal, Bw, 0, open)$
  $gLim(Bw) \leftarrow gLim(Fw) \leftarrow 0$
  $incrementedDir \leftarrow Bw$
  for gSum from 1 up by 1 until unsolvable do
    $incrementedDir == opposite(incrementedDir) + 1$
    if expandLevel(nodes,gLim(),gSum,H) then
      $return(gSum)$
    end if
  end for
```



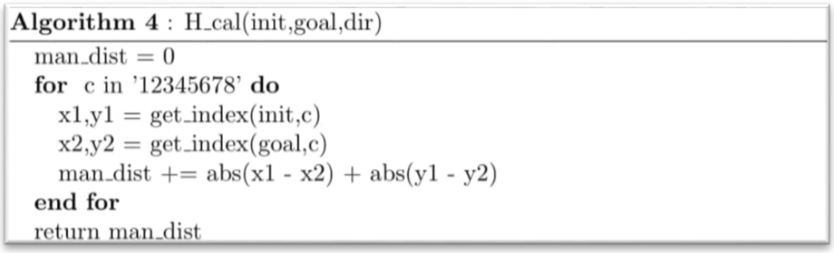
Algorithm 2 : $expandLevel(nodes, gLim(), gSum(), H) \rightarrow solved$

```
open = $n | n \epsilon nodes \wedge expandableThisLevel(n)$
while $open \neq \{\}$ do
  $n \leftarrow Pop(n \epsilon open \wedge gSum >= H(n, dir(n)) + g(n))$
  if $expandableThisLevel(n) == False$ then
    $remove\ n\ from\ open\ list$
  else
    continue
  end if
  $status(n) \leftarrow' closed'$
  for all $neighbour\ in\ expand(n, dir(n))$ do
    $child \leftarrow (neighbour, dir(n), g + 1, open)$
    if $neighbour\ already\ exist\ in\ nodes$ then
      $node = get(neighbour, nodes)$
      if $dir(child) == dir(node)$ then
        $continue$
      else
        $return(True)$
      end if
    end if
    if $expandableThisLevel(child)$ then
      $open+ = child$
    end if
  end for
end while
return(False)
```



Algorithm 3 : H(init,current$_n ode, dir$)

```
for node in nodes do
  if $getdirection(node)! = dir \wedge state(node) ==' open'$ then
    $add\ node\ to\ opposite\ nodes\ list$
  end if
end for
for node in opposite nodes list do
  goal = node
  $calculate\ h\ minimum = h\_cal(current\_node, goal, dir) + gvalue(node)$
end for
$return\ minimum\_\ h$
```



Algorithm 4 : H_cal(init,goal,dir)

```
man_dist = 0
for c in '12345678' do
  x1,y1 = get_index(init,c)
  x2,y2 = get_index(goal,c)
  man_dist += abs(x1 - x2) + abs(y1 - y2)
end for
return man_dist
```

Fig. 3. Bidirectional Front-Front Search using Manhattan Distance

Approach 4: Bidirectional Front-End Search using Abstraction

Pseudocode for Bidirectional Front-End using Abstraction is given in figure 4. It uses the same algorithm as approach 2 except now instead of using Manhattan distance we use abstraction as our heuristic function. The concept of abstraction was inspired by (Faster Optimal and Suboptimal Hierarchical Search, 2011) which was discussed before in related research section. As shown in Algorithm 3 of figure 4, we map our current node (node which we are currently at), goal node and start node (init) from base space to an abstract space where we reduce the complexity of their states. In addition, to compute heuristic we apply standard blind search (Approach 1 algorithm) in the abstract space which will return the optimal solution cost path from the abstracted node to goal/start depending on the direction of the current node.

```
Algorithm 1 : $B\_F2E\_Abstraction(init, goal, H) \rightarrow optimalSolutionCost$
  if already solved then
    $return(0)$
  end if
  $nodes \leftarrow (init, Fw, 0, open), (goal, Bw, 0, open)$
  $gLim(Bw) \leftarrow gLim(Fw) \leftarrow 0$
  $incrementedDir \leftarrow Bw$
  for gSum from 1 up by 1 until unsolvable do
    $incrementedDir == opposite(incrementedDir) + 1$
    if expandLevel(nodes,gLim(),gSum,H) then
      $return(gSum)$
    end if
  end for
```

```
Algorithm 2 : $expandLevel(nodes, gLim(), gSum(), H) \rightarrow solved$
  $open = n | n \epsilon\ nodes \wedge\ expandableThisLevel(n)$
  while $open \neq \{\}$ do
    $n \leftarrow Pop(n\ \epsilon\ open\ \wedge\ gSum >= H(n, dir(n)) + g(n))$
    $status(n) \leftarrow' closed'$
    for all $neighbour\ in\ expand(n, dir(n))$ do
      $child \leftarrow (neighbour, dir(n), g + 1, open)$
      if $neighbour\ already\ exist\ in\ nodes$ then
        $node = get(neighbour, nodes)$
        if $dir(child) == dir(node)$ then
          $continue$
        else
          $return(True)$
        end if
      end if
      if $expandableThisLevel(child)$ then
        $open+ = child$
      end if
    end for
  end while
  return(False)
```

```
Algorithm 3 : H_abstraction(init,current_node,dir)
  $goal = ['1','1','1','1','5','6','7','8','0']$
  $map(init\ node)\ \rightarrow\ abstract\ node$
  $map(current\ node)\ \rightarrow\ abstract\ node$
  if $dir ==' fw'$ then
    $h = blind(current\_node, goal)$
    $return\ h$
  else
    $h = blind(current\_node, init)$
    $return\ h$
  end if
```

Approach 5: Bidirectional Front-Front Search using Abstraction

Pseudocode for Bidirectional Front-Front using abstraction is given in Figure 5. Algorithm 1, 2 and 3 are like our previous approaches. We introduce 3 phases in the abstract space to find the heuristic value of the node in the base space. In abstract phase 1, we expand all the nodes in the opposite frontier until we exhaust all nodes (i.e. each node's gvalue = max (gvalue of all the nodes in the opposite frontier)) and check if we find a collision with our current node. If we don't find a collision, abstract phase 2 is called where we expand all neighbors of our current node until we exhaust all nodes (i.e. creating a frontier similar to phase 1), also we keep checking for a collision. If we still can't find a collision, abstract phase 3 is called, which is simply a blind search we run on the two frontiers we got from our previous two phases. Algorithm 4, 5 and 6 of Figure 5 provides detailed steps for all 3 phases.

```
**Algorithm 1** : $B\_F2F\_Abstraction(init, goal, H) \rightarrow optimalSolutionCost$
  **if** already solved **then**
    $return(0)$
  **end if**
  $nodes \leftarrow (init, Fw, 0, open), (goal, Bw, 0, open)$
  $gLim(Bw) \leftarrow gLim(Fw) \leftarrow 0$
  $incrementedDir \leftarrow Bw$
  **for** gSum from 1 up by 1 until unsolvable **do**
    $incrementedDir == opposite(incrementedDir) + 1$
    **if** expandLevel(nodes,gLim(),gSum,H) **then**
      $return(gSum)$
    **end if**
  **end for**
```

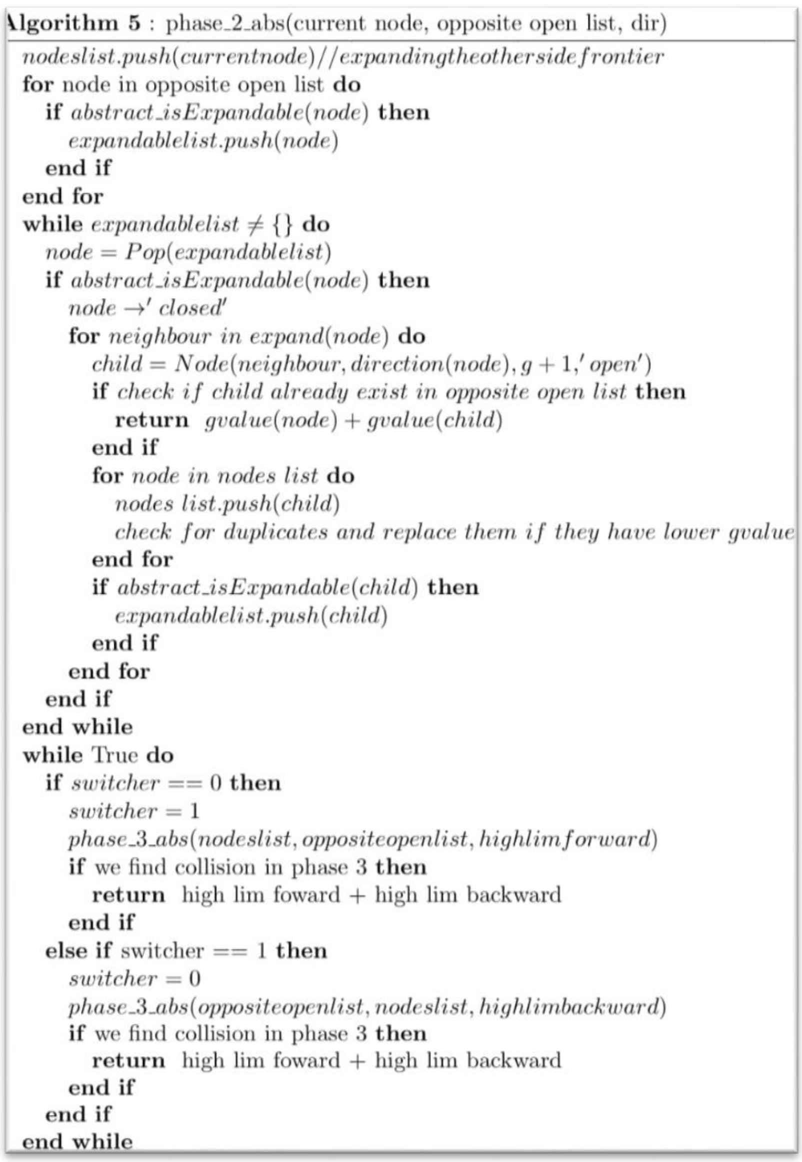
```
**Algorithm 2** : $expandLevel(nodes, gLim(), gSum(), H) \rightarrow solved$
  $open = n|n \epsilon nodes \wedge expandableThisLevel(n)$
  **while** $open \neq \{\}$ **do**
    $n \leftarrow Pop(n \epsilon open \wedge gSum >= H(n, dir(n)) + g(n))$
    $status(n) \leftarrow' closed'$
    **for all** $neighbour$ $in$ $expand(n, dir(n))$ **do**
      $child \leftarrow (neighbour, dir(n), g + 1, open)$
      **if** $neighbour$ $already$ $exist$ $in$ $nodes$ **then**
        $node = get(neighbour, nodes)$
        **if** $dir(child) == dir(node)$ **then**
          $continue$
        **else**
          $return(True)$
        **end if**
      **end if**
      **if** $expandableThisLevel(child)$ **then**
        $open+ = child$
      **end if**
    **end for**
  **end while**
  return(False)
```

Algorithm 3 : $H_abstraction(init, current_node, dir)$

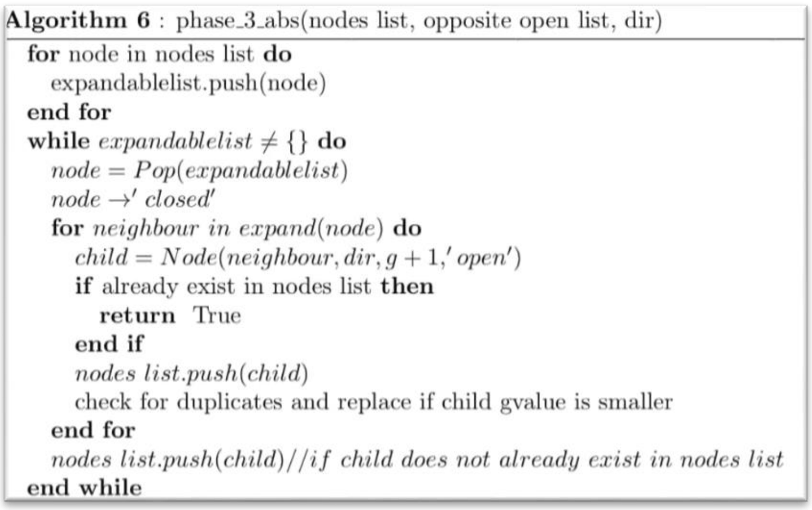
```
for node in nodes do
  if $getdirection(node)! = dir \wedge state(node) ==' open'$ then
    $add\ node\ to\ opposite\ nodes\ list$
  end if
end for
$goal = ['1','1','1','1','5','6','7','8','0']$
$map(init) \rightarrow abstractnode\ (i.e\ Tiles\ 1,2,3,4 \rightarrow\ 1)$
$map(currentnode)\ \rightarrow\ abstract\ node$
for node in opposite open list do
  $map(node)\ \rightarrow\ abstract\ node$
end for
phase_1 = phase_1 _abs(current node, opposite open list, dir)
return phase_1
```

Algorithm 4 : phase_1_abs(current node, opposite open list, dir)

```
$high\_lim\ =\ max(gvalue(node\ in\ opposite\ open\ list))$
for node in opposite open list do
  if $abstract\_isExpandable(node)$ then
    $expandablelist.push(node)$
  end if
end for
while $expandablelist \neq \{\}$ do
  $node = Pop(expandablelist)$
  if $abstract\_isExpandable(node)$ then
    $node \rightarrow' closed'$
    for $neighbour\ in\ expand(node)$ do
      $child = Node(neighbour, direction(node), g+1,' open')$
      if $neighbour == currentnode$ then
        $return\ gvalue(child)$
      end if
      for $node\ in\ opposite\ open\ list$ do
        $opposite\ nodes\ list.push(child)$
        $check\ for\ duplicates\ and\ replace\ them\ if\ they\ have\ lower\ gvalu$
      end for
      if $abstract\_isExpandable(child)$ then
        $expandablelist.push(child)$
      end if
    end for
  end if
end while
return phase_2_abs(current node,opposite open list,high_lim,current dir)
```



```
Algorithm 5 : phase_2_abs(current node, opposite open list, dir)
nodeslist.push(currentnode)//expandingtheothersidefrontier
for node in opposite open list do
  if abstract_isExpandable(node) then
    expandablelist.push(node)
  end if
end for
while expandablelist $\neq \{\}$ do
  node = Pop(expandablelist)
  if abstract_isExpandable(node) then
    node $\rightarrow'$ closed'
    for neighbour in expand(node) do
      child = Node(neighbour, direction(node), $g + 1$,' open')
      if check if child already exist in opposite open list then
        return gvalue(node) + gvalue(child)
      end if
      for node in nodes list do
        nodes list.push(child)
        check for duplicates and replace them if they have lower gvalue
      end for
      if abstract_isExpandable(child) then
        expandablelist.push(child)
      end if
    end for
  end if
end while
while True do
  if switcher == 0 then
    switcher = 1
    phase_3_abs(nodeslist, oppositeopenlist, highlimforward)
    if we find collision in phase 3 then
      return high lim foward + high lim backward
    end if
  else if switcher == 1 then
    switcher = 0
    phase_3_abs(oppositeopenlist, nodeslist, highlimbackward)
    if we find collision in phase 3 then
      return high lim foward + high lim backward
    end if
  end if
end while
```



```
Algorithm 6 : phase_3_abs(nodes list, opposite open list, dir)
for node in nodes list do
  expandablelist.push(node)
end for
while expandablelist $\neq \{\}$ do
  node = Pop(expandablelist)
  node $\rightarrow'$ closed'
  for neighbour in expand(node) do
    child = Node(neighbour, dir, $g + 1$,' open')
    if already exist in nodes list then
      return True
    end if
    nodes list.push(child)
    check for duplicates and replace if child gvalue is smaller
  end for
  nodes list.push(child)//if child does not already exist in nodes list
end while
```

Fig. 5. Pseudocode for Bidirectional Front-Front using Abstraction

4. Analysis

Scatter graphs shown in Figure 6, 6.1 and 6.2 shows the number of nodes expanded for each optimal solution cost level for Bidirectional Blind, Front-End using Manhattan and Front-Front using Manhattan respectively. As we can see, Front-Front expands fewer nodes than Front-End when Manhattan heuristic is used. Furthermore, as expected our naïve approach i.e. Bidirectional blind search overwhelms both Front-Front and Front-End approaches. Also, we define our domain to be 8-Puzzle and limit of our problem set to 100 problems per solution path (with solution cost ≤ 21).

On the other hand, scatter graphs shown in Figure 6.3 and 6.5 shows the number of nodes expanded in base space for each optimal solution cost level (optimal solution cost ≤ 21) for Front-End and Front-Front using Abstraction respectively. We can notice that Front-Front expands slightly more nodes than Front-End (in base space). Whereas, in Figure 6.4 and 6.6 we can clearly see that Front-End expands much more nodes than Front-Front in the abstract space ($18 \leq$ optimal solution cost ≤ 21).

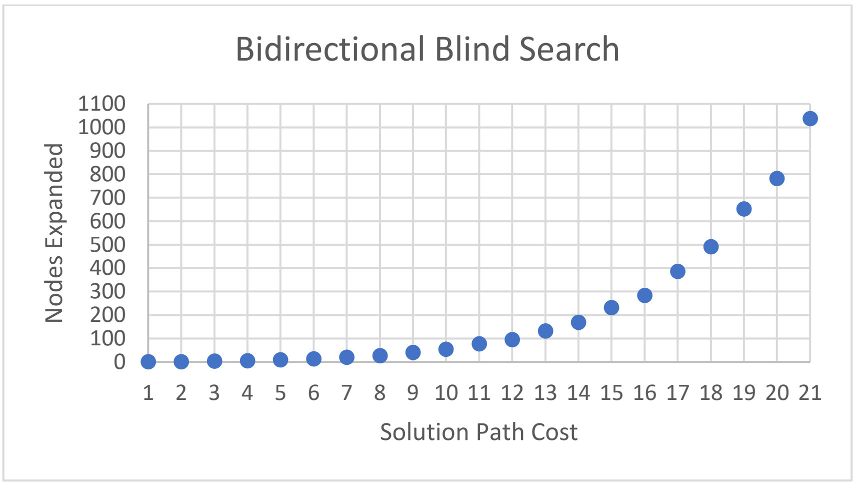


Fig. 6

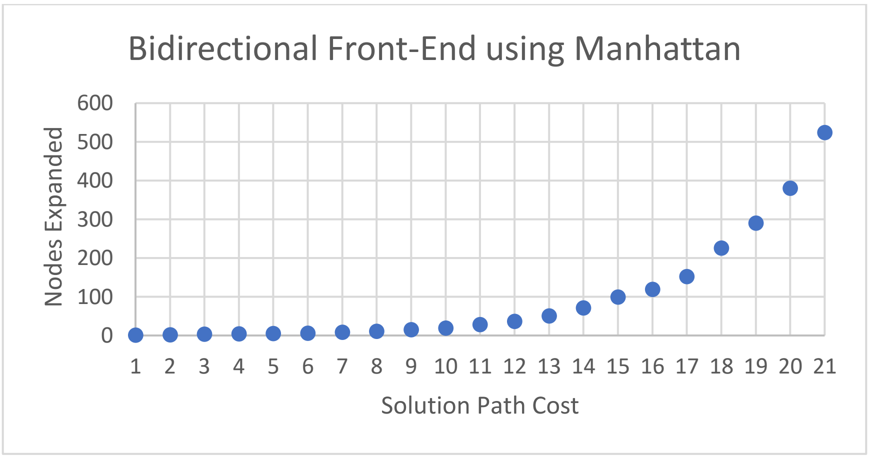


Fig. 6.1

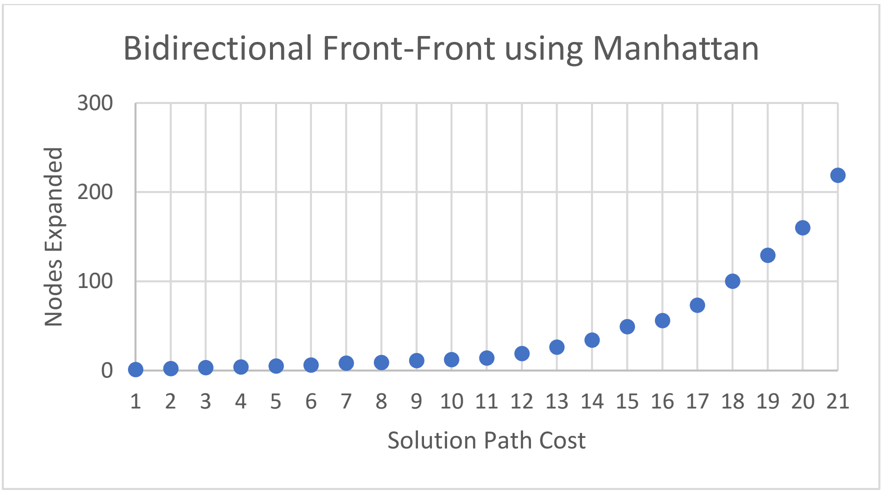


Fig. 6.2

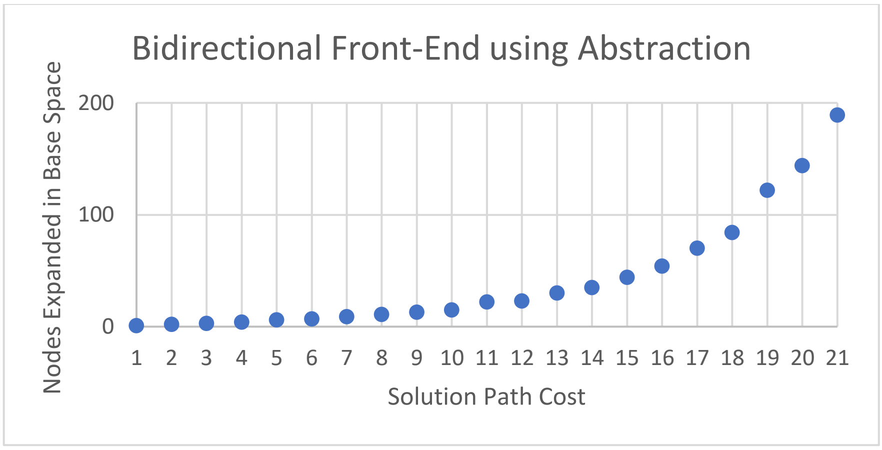


Fig. 6.3

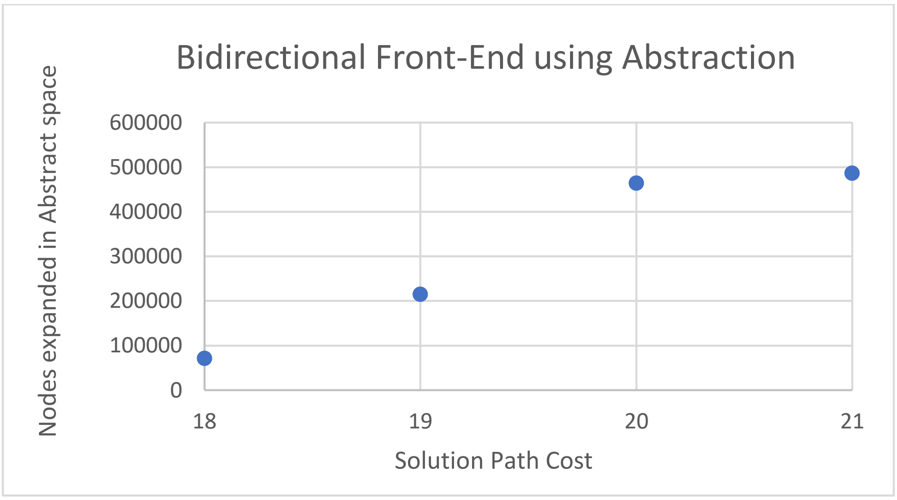


Fig. 6.4

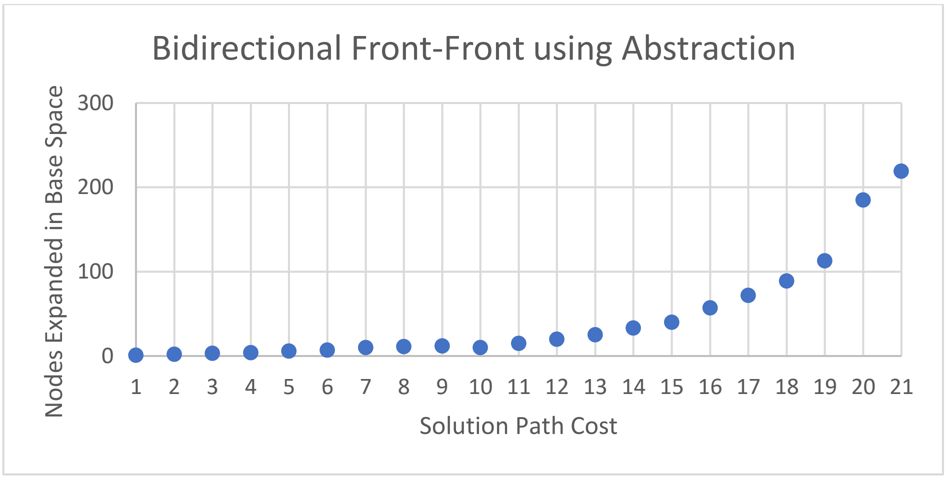


Fig. 6.5

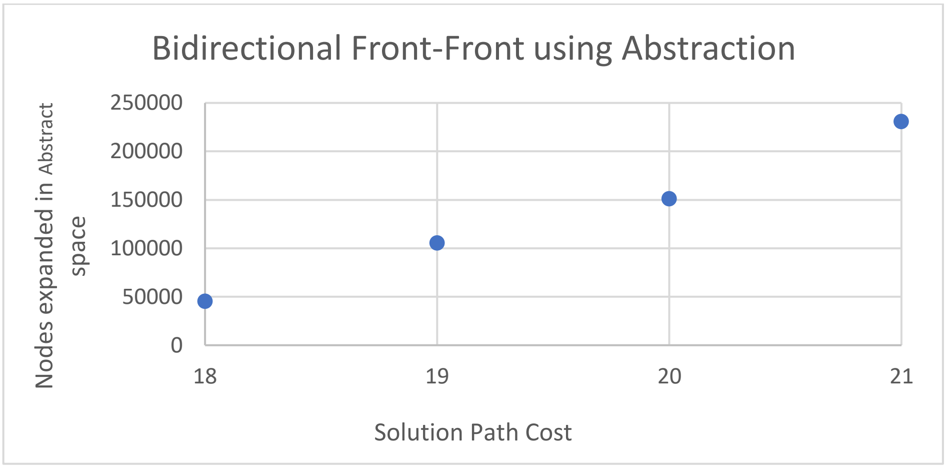


Fig. 6.6

Furthermore, graphs in Figure 7, 7.1 and 7.2 shows average runtime-per-node (Note: Time is in microseconds) to find the optimal solution (18 ≤ solution path cost ≤ 21, with an average of 50 problems per solution path cost) for Blind, Front-End using Manhattan and Front-Front using Manhattan respectively. As expected, Front-Front algorithm's runtime-per-node is much higher than other two algorithms. Moreover, Front-End using Manhattan is quicker than Blind in terms of total runtime (runtime-per-node x number of nodes expanded).

On the other hand, we have graphs in Figure 7.4 and 7.5 which show average runtime-per-node (Note: Time is in Milliseconds) for Front-End and Front-Front using Abstraction respectively. We can see that Front-End overwhelmingly outruns Front-Front in terms of runtime-per-node and total runtime.

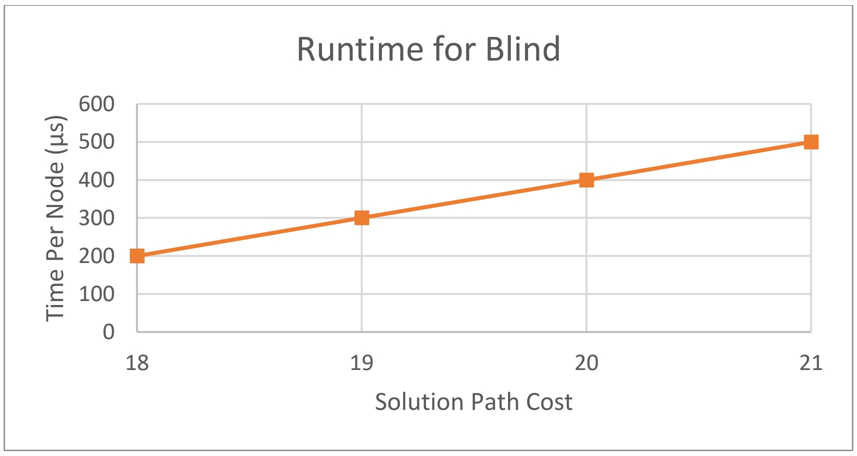


Fig. 7

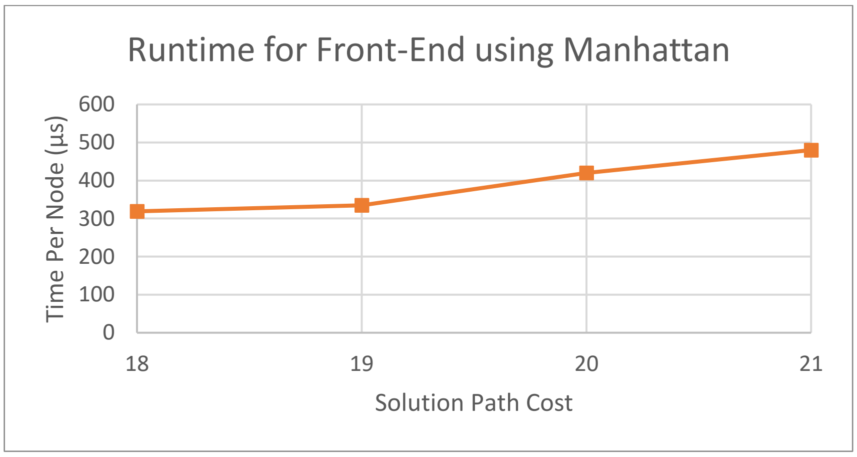


Fig. 7.1

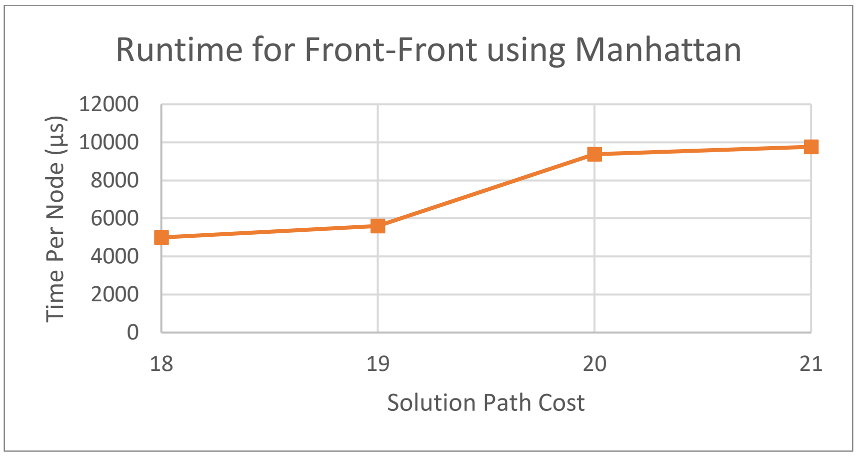


Fig. 7.2

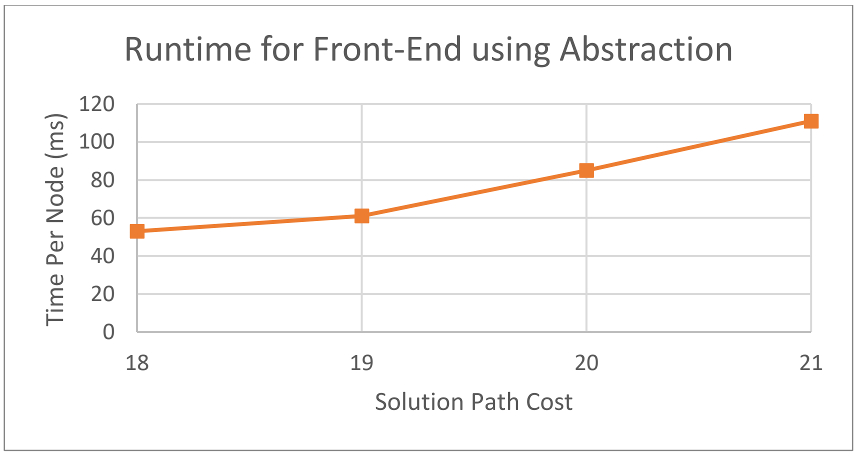


Fig. 7.3

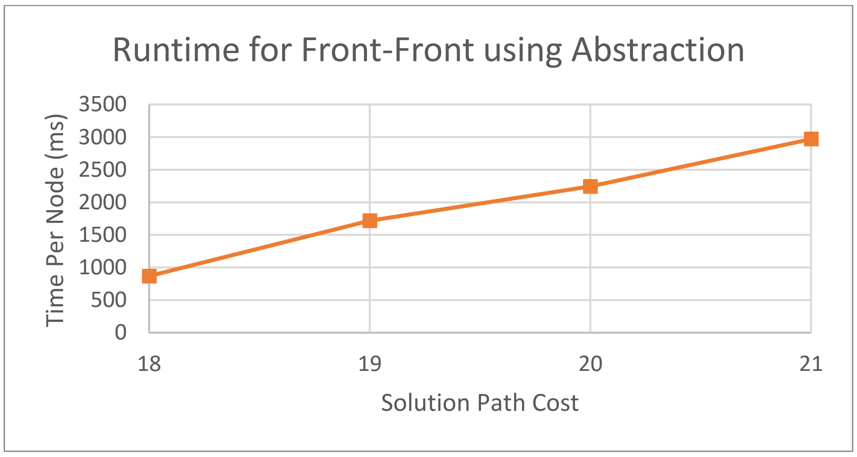


Fig. 7.4

5. Summary

To conclude, this research shows that the Bidirectional Front-End using Manhattan approach outruns Front-Front using Manhattan and Blind in terms of runtime whereas Bidirectional Front-Front using Manhattan outperforms Blind and Front-End using Manhattan in terms of the number of nodes expanded. Furthermore, Bidirectional Front-Front does expand much less node (base space + abstract space) than Front-End when Abstraction is used, Bidirectional Front-Front using Abstraction does show promising results in terms of the number of nodes expanded but it fails to produce solution quickly. Finally, this research can be further extended if I get to work on it for another 3-6 months by analyzing different domains, improving current algorithm by trying out different caching techniques, usage of pattern database and/or adding the concept of switchback discussed in (Searching Without a Heuristic: Efficient Use of Abstraction, 2010).

6. References

D. de Champeaux, L. Sint. An Optimality Theorem for a Bi-Directional Search Algorithm.

Bradford Larsen, Ethan Burns, Wheeler Ruml, Robert C. Holte. 2010. Searching Without a Heuristic: Efficient Use of Abstraction.

Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical Heuristic Search Revisited.

Michael J. Leighton, Wheeler Ruml, Robert C. Holte. 2011. Faster Optimal and Suboptimal Hierarchical Search.