Combining Speedup Techniques based on Landmarks and Containers

R. KALPANA
Department of Computer Science & Engineering
Pondicherry Engineering College
Pondicherry, India
rkalpana@pec.edu

Dr. P. THAMBIDURAI
Perunthalaivar Kamarajar Institute of Engineering & Technology
Karaikal, Pondicherry
India

***I. Abstract*—The Dijkstra's algorithm [1] , which is applied in many real world problems like mobile routing, road maps, railway networks, etc,. is used to find the shortest path between source and destination. There are many techniques available to speedup the algorithm while guaranteeing the optimality of the solution. The main focus of the work is to implement landmark technique and Containers separately and compare the results with random graphs and planar graphs. The combined speedup technique which is based on landmarks and containers were also experimented with random graphs and planar graphs to improve the speedup of the shortest path queries.**

Keywords- Dijkstra's Algorithm, Graph Theory, Land mark technique, Geometric Containers , speed-up

I. INTRODUCTION

A directed simple graph G is a pair (V, E), where V is the set of nodes / vertices and $E \subseteq V \times V$ is a set of edges, where an edge is an ordered pair of nodes of the form (u, v) such that $u, v \in V$. Usually the number of nodes $|V|$ is denoted by n and the number of edges $|E|$ is denoted by m. A path in graph G is a sequence of nodes $(u_1,..., u_k)$ so that $(u_i, u_{i+1}) \in E$ for all $1 \le i < k$. A path in which $u_1 = u_k$ is called a cycle or cyclic path.Given the edge weights $l : E \to R$, the length of the path $P = (u_1,..., u_k)$ is the sum of the lengths of its edges $l(P) := \sum_{1 \le i < k} l(u_i, u_{i+1})$. For any two nodes $s, t \in V$, a shortest s-t path is a path of minimal length with $u_1 = s$ and $u_k = t$. The distance $d(s,t)$ between s and t is the length of the shortest path s-t. A layout of a graph $G = (V, E)$ is a function $L : V \to R^2$ that assigns each node a position in R^2. A graph is called sparse if $m = O(n)$.

Let G = (V, E) be a directed graph whose edges are weighted by a function $w : E \to R$. The weights are interpreted as the edges' or lengths in the sense that the length of a path is the sum of the weights of its edges. The single-source single-target (SSST) shortest-path problem consists in finding a path of minimum length from a given source $s \in V$ to a given target $t \in V$. The problem is only well defined for all pairs, if G does not contain negative cycles. In the presence of negative weights, but not negative cycles, it is possible, using Johnson's algorithm, to convert in $O(nm + n^2 \log n)$ time the original edge weights $w : E \to R$ to non-negative edge weights $w' : E \to R_0^+$ that result in the same shortest paths. Hence, it can be safely assumed that the edge weights are non-negative. It can also be assumed that for all pairs $(s,t) \in V \times V$, the shortest path from *s* to *t* is unique.

The classical algorithm for computing shortest paths in a directed graph with nonnegative edge weights is that of Dijkstra's algorithm. Dijkstra's algorithm implemented with Fibonacci heaps is still the fastest known algorithm for the general case of arbitrary nonnegative edge lengths, taking $O(m + n \log n)$ worst-case time. For special cases (eg. undirected graphs, integral or uniformly distributed edge weights), better algorithms are identified.

II. RELATED WORK

A. *Basic speedup technique*

Computing shortest paths between nodes in a given directed graph is classically solved by Dijkstra's algorithm[1]. But besides Dijkstra's algorithm there are many recent algorithms that solve variants and special cases of the shortest-path problem with better running time. This section also focuses on variants of Dijkstra's algorithm (also denoted as speedup techniques in the following) that further exploit the fact that a target is given. Typically, such improvements of Dijkstra's algorithm cannot be proved to be asymptotically faster than the original algorithm, and, in this sense are heuristics. However, it can be empirically shown that they indeed improve the running time drastically for many realistic data sets. An overview of the speedup techniques is as follows

- *Goal-directed search:* The given edge weights are modified to favour edges leading toward the target node [2]. With graphs from timetable information, a speed-up in running time of a factor of roughly 1.5 is reported [2]
- *Bidirectional search:* Start a second search backward, from the target to the source[3]. Both searches stop

when their search horizons meet. Using bidirectional search space can be reduced by a factor of 2.

- *Multilevel approach:* This approach takes advantage of hierarchical coarsening of the given graph, where additional edges have to be computed. These edges can be regarded as distributed to multiple levels. Depending on the given query, only a small fraction of these edges have to be considered to find a shortest path. Using this technique, speed-up factors of more than 3.5 were observed for road map and public transport graphs [4]. Timetable information queries could be improved by a factor of 11 [5].

- *Shortest-path containers:* These containers[6] provide a necessary condition for each edge, whether or not it has to be respected during the search. More precisely, the set of all nodes that can be reached on a shortest path using this edge is stored. Speedup factors in the range between 10 and 20 can be achieved.

B. Combining Speedup Techniques

Combining each pair of techniques is outlined in [7] and it is noted that extending these to combinations, including three or all four techniques, are not difficult.

Goal-Directed Search and Bidirectional Search: Combining goal-directed and bidirectional search[7] is not as obvious as it may seem. Simple application of a goal-directed search forward and backward yields a wrong termination condition. In certain situations the search in each direction almost reaches the sources of the other direction. This often results in a slower algorithm.

To overcome these deficiencies, it is preferable to use the very same edge weights $l'(v,w) = l(v,w) - \lambda(v) + \lambda(w)$ for both the forward and the backward search. With these weights, the forward search is directed to the target *t* and the backward search has no preferred direction, but favours edges that are directed towards *t*. This proceeding always computes shortest paths, as an *s-t* path is shortest independent of whether l or l' is used for the edge weights.

Goal-Directed Search and Multilevel Approach: The multilevel pproach determines, for each query, a subgraph of the multilevel graph on which Dijikstra's algorithm is finally run. The computation of this subgraph does not affect edge lengths and thus a goal-directed search can be simply performed on it.

Goal-Directed Search and Shortest-Path Containers: Similar to the multilevel approach, the shortest-path containers approach determines for a given query a subgraph of the original graph. Again, edge lengths are irrelevant for the computation of the subgraph and goal-directed search can be applied readily.

Bidirectional Search and Multilevel Approach: A bidirectional search can be applied to the subgraph defined by the multilevel approach. The subgraph can be computed on the fly during Dijikstra's algorithm: for each node considered, the set of necessary outgoing edges is determined. To perform a bidirectional search on the multilevel subgraph, a symmetric, backward version of the subgraph computation has to be implemented: for each node considered in the backward search, the incoming edges that are part of the subgraph have to be determined. Shortest paths are guaranteed, since bidirectional search is run on a subgraph that preserves optimality, and, by the additional edges, only contains supplementary information consistent with the original graph.

Bidirectional Search and Shortest-Path Containers: In order to take advantage of shortest-path containers in both directions of a bidirectional search a second set of containers is needed. For each edge $e \in E$, the set $S_b(e)$ is computed containing those nodes from which a shortest path ending with e exists. For each edge $e \in E$ the bounding box of $S_b(e)$ is stored in an associative array C_b with index set E. The forward search checks whether the target is contained in C(e), the backward search, checks whether the source is in $C_b(e)$. It can be verified that by construction only such edges are pruned that do not form part of any partial shortest path and thus of any shortest *s-t* path.

Multilevel Approach and Shortest-Path containers: The multilevel approach enriches a given graph with additional edges. Each new edge (u_1, u_k) represents a shortest path $(u_1,u_2,...,u_k)$ in *G*. Such a new edge (u_1, u_k) is annotated with $C(u_1,u_2)$, the associated bounding box of the first edge on this path. This consistent labelling of new edges, which represent shortcuts in the original graph, ensures still shortest paths.

Hierarchical and Goal-directed speed-up techniques. The combination of hierarchical and goal directed speedup techniques [8], [9] found to give best results for unit disk graphs, grid networks, and time-expanded timetables. It is suggested that the goal directed technique can be applied to higher levels of hierarchy.

C. A* Search and Landmarks

In this section a new shortest path algorithm that uses A* search in combination with a new graph-theoretic lower-bounding technique based on landmarks and the triangle inequality is explained[2]. The algorithm computes optimal shortest paths and works on any directed graph. Experimental results show that the new technique outperforms A* search with Euclidean bounds, by a wide margin on road networks [2].

Potential Function

A potential function is a function, from vertices to reals. Given a potential function π, the reduced cost of an edge is defined as follow

$$l_{\pi}(v,w) = l(v,w) - \pi(v) + \pi(w) \qquad (2.1)$$

Suppose l is replaced by l_{π} then for any two vertices x and y, the length of any x-y path changes by the same amount $\pi(y) - \pi(x)$. Thus a path is a shortest path with respect to l iff it is a shortest path with respect to l_π and the two problems are equivalent. Note that, π is feasible if l_π is nonnegative for all arcs. It is well known that if $\pi(t) \le 0$ and π is feasible, then for any v, $\pi(v)$ is a lower bound on the distance from v to t. It is to be noted that

Lemma 2.1 If π_1 and π_2 are feasible potential functions, then $p = \max(\pi_1, \pi_2)$ is feasible.

Feasible potential functions can be combined by taking the minimum, or, the average of feasible potential functions. The maximum is used in particular to combine feasible lower bound functions in order to get one that any vertex is at least as high as each original function.

A* Search

Consider the problem of looking for a path from s to t and assume that a function $\pi_t : V \to R$ exists such that $\pi_t(v)$ gives an estimate on the distance from v to t. A* search is an algorithm that works like Dijkstra's algorithm, except that at each step it selects a labelled vertex v with the smallest value $k(v) = d_s(v) + \pi_t(v)$ to scan next. It is easy to verify that A* search is equivalent to Dijkstra's algorithm on the graph with length function l_{π_t} .

D. Geometric containers for efficient shortest path computation

A fundamental approach in finding efficiently best routes or optimal itineraries in traffic information is to reduce the search space of the most commonly used shortest path routine (Dijikstra's algorithm) on a suitable defined graph. Reduction of the search space should simultaneously be combined with ways of retaining data structures, created during a preprocessing phase of size linear to the size of the graph. The search space of Dijikstra's algorithm can be significantly reduced by extracting geometric information from a given layout of the graph and by encapsulating precomputed shortest-path information in resulted geometric objects (containers) [6]. When edge weights are subject to change, methods exist for dynamically updating the containers instead of recomputing everything from scratch [6].

1) Shortest-Path Containers: In this section, we consider the concept of containers, which helps to reduce the search space of Dijkstra's algorithm. Containers are used to keep the nodes, which are potentially useful for shortest-path computations. This idea gives rise to Dijkstra's Algorithm with Pruning[6], which reduces the search space by examining, at each iteration, only a subset of the neighbors of a node (line 5a); the differences to Dijkstra's algorithm are shown in boldface. The condition in line 5a is formalized by the notion of a consistent container.

Definition 2.1. Let G = (V, E), w : E → R be a weighted graph. A set of nodes C ⊆ V is called a container. A container C associated with an edge (u, v) is called consistent, if for all shortest paths from u to t that start with the edge (u, v), the target t is in C.

In other words, C(u, v) is consistent, if S(u, v) ⊆ C(u, v), where S(u, v) represents the set of nodes x for which the shortest u-x-path starts with the edge (u, v). Note that further nodes may be part of a consistent container. However, at least the nodes that can be reached by a shortest path starting with (u, v) must be in C(u, v). The additional nodes are referred as wrong nodes, since they lead the search in the wrong way.

Theorem 2.1. Let G = (V, E), w: E → R be a weighted graph and for each edge e let C(e) be a consistent container. Then, Dijkstra's Algorithm with Pruning finds a shortest path from s to t.

Proof. Consider the shortest path P from s to t that is found by Dijkstra's algorithm. If for all edges e ∈ P the target node t is in C(e), the path P is found by Dijkstra's Algorithm with Pruning, because the pruning does not change the order in which the edges are processed. A subpath of a shortest path is again a shortest path, so for all (u, v) ∈ P, the subpath of P from u to t is a shortest u-t-path. Then, by the definition of consistent container, t ∈ C(u, v).

Definition 2.2. Let C denote a set of containers and for each edge e ∈ E let S (e) ⊆ V denote the set of nodes that can be reached by a shortest path starting with e. For both sets, the number of nodes inside all containers is counted: $\sum_{e \in E} \left|\{t \in C(e)\}\right|$ and $\sum_{e \in E} \left|\{t \in S(e)\}\right|$. Both sums are bounded by n · m. Therefore the quality of C can be defined as:

$$\frac{n \bullet m - \sum_{e \in E} \left|\{t \in C(e)\}\right|}{n \bullet m - \sum_{e \in E} \left|\{t \in S(e)\}\right|}$$

This fraction is biased by the number of correct nodes. It equals 1, if the number of wrong nodes inside containers is zero, while it becomes 0, if all containers in C contain the entire graph.

III. COMBINING SPEEDUP TECHNIQUES

A. LANDMARKS:

The search space of Dijkstra's algorithm can be reduced by using landmarks. Heuristic estimates on the distance of a vertex to the target can be calculated using landmarks. Landmarks tend to attract the search towards them and so by appropriately selecting landmarks the overall performance can be improved.

The procedure in Algorithm 1 outlines the shortest path computation technique with heuristic values modifying the priority of vertices. Lines 4a and 6 are the changes made to the original Dijkstra's algorithm. The purpose of line 4a is evident of its own because the problem under consideration is single source single target shortest path problem. The key change is that of line 6. Traditional Dijkstra's algorithm considers only the distance of a vertex from the source whereas in Algorithm 1 potential (u) is used an estimate of the distance from the vertex to the target. So a heuristic / potential function can direct the search towards the target thereby reducing the search space considerably.

```
1   for all nodes u belongs to V
      set dist(u) := infinity
2   initialize priority queue Q with source s and dist(s) := 0
3   while priority queue Q is not empty
4     get node u with smallest tentative distance dist(u) in Q
4a    if u = t return
5     for all neighbor nodes v of u
6         set new-dist := dist(u) + w(u, v) + potential(u)
7         if new-dist < dist(v)
8             if dist(v) = infinity
9               insert neighbor node v in Q with priority new-dist
10            else
11              set priority of neighbor node v in Q to new-dist
12            set dist(v) := new-dist
```

Algorithm 1. Using Landmarks in Shortest Path Computation

Landmark Selection: For incorporating landmarks into shortest path computation the following additions are to be made to the existing path computation technique: A procedure for selecting landmarks, computing the distance values from the landmarks to the remaining vertices and utilizing the computed distance values to obtain heuristic estimates which could be used to modify the priority of vertices to be considered.

```
for(int landmarkSelected = 0; landmarkSelected < LANDMARKCOUNT;
      landmarkSelected++) {
  forall_nodes(v,G) {
    if(v == s) dist[v] = 0; else dist[v] = MAXDOUBLE;
    PQ.insert(v,dist[v]);
  }
  while ( !PQ.empty() ) {
    u = PQ.del_min();
    if( dist[u] == MAXDOUBLE ) {
        PQ.clear();
        break;
    }
    forall_adj_edges(e,u) {
      v = target(e);

      double c = dist[u] + cost[e];
      if ( c < dist[v] ) {
        PQ.decrease_p(v,c);  dist[v] = c;
      }
    }//Neighbour distance updation ends
  }//While the Priority Queue has vertices to be explored
  landmarks.append( u ); //Select the farthest node from s as landmark
  s = u; //Next Source for another landmark selection
} //Landmark Selection loop
```

Code Segment 1. Farthest Landmark Selection Technique

The landmark selection procedure is briefed in code segment 1. The procedure used for selecting landmarks is called "Farthest landmark selection technique". The idea behind this procedure is that, landmarks are chosen in such a way that they are far apart, i.e. the landmarks are spread throughout the entire graph and this helps to obtain good potential values for any vertex chosen at random without any bias. The data structure "landmarks" is a list containing the landmarks chosen by the procedure. "LANDMARKCOUNT" indicates the number of landmarks required.

The selection procedure proceeds as follows. A single source all target shortest path query is initiated similar to traditional Dijkstra's algorithm. The vertices deleted from the priority queue are kept track of and the final vertex to be deleted from the queue is added to the list of landmarks. The final vertex is selected as a landmark because in Dijkstra's algorithm the vertices are always considered in the increasing order of their shortest path distance and the final vertex deleted from the queue is the one farthest from the source. The selection procedure is repeated with the newly selected landmark as the source. Once the required number of landmarks are selected the procedure stops.

1) Calculating and Using Heuristic Values:

```
//Include Landmark based potentials also
maxdiff = nodeLandmarkInfo[v].nLInfo[0].dist –
                  nodeLandmarkInfo[t].nLInfo[0].dist;

for(int landmarkCount = 1; landmarkCount < LANDMARKCOUNT;
        landmarkCount++) {
     diff = nodeLandmarkInfo[v].nLInfo[landmarkCount].dist –
nodeLandmarkInfo[t].nLInfo[landmarkCount].dist;
               //Triangle Inequality part

     if( diff > maxdiff )
           maxdiff = diff; //Choose the max Lower Bound
}

double c = dist[u] + cost[e] + maxdiff; //Update cost with heuristic value
```

Code Segment 2. Updating cost using Landmark based heuristic values

The distance from landmarks to the remaining vertices should be calculated for obtaining potential values. The distance calculation requires initiating a single source all target shortest path computation from each of the landmarks. The obtained values are stored as follows. "nodeLandmarkInfo" is a vertex array, containing an array of landmark and corresponding distance values. So the distance between a vertex 'v' and the i^{th} landmark can be accessed as nodeLandmarkInfo[v].nLInfo[i].dist.

Code Segment 2. highlights the modifications to be made during the actual shortest path evaluation. Before updating the distance value of a vertex 'v', the maximum "distance difference" between the vertex 'v' and the various landmarks is calculated and stored in "maxdiff". This serves as a heuristic estimate of the distance between the vertex and the target. Hence the priority is updated only if the sum of, known distance from source and an estimate of the distance from the vertex to the target is less than the previously available priority.

B. SHORTEST PATH CONTAINERS:

The Geometric containers help to reduce the search space of Dijkstra's algorithm by enclosing a list of target nodes for each edge inside a geometric object. The geometric information associated with each edge is then used for improving the performance of shortest path computations. Let G= (V, E), w: E → R be a weighted graph. It is remembered that a set of nodes C⊆ V is called a container. A container C associated with an edge (u, v) is called consistent, if for all shortest paths from u to t that start with the edge (u, v), the target t is in C. In other words, C (u, v) is consistent, if S (u, v) ⊆ C (u, v), where S (u, v) represents the set of nodes x for which the shortest u-x-path starts with the edge (u, v). Note that further nodes may be part of a consistent container. However, at least the nodes that can be reached by a shortest path starting with (u, v) must be in C (u, v). The additional nodes are referred as wrong nodes, since they lead the search in the wrong way.

1) Creating Consistent Containers: S(e) is the set of all nodes t with the property that there is a unique shortest s-t path that starts with the edge e. To determine S (s, x) for every edge (s, x) ∈ E, dijkstra's algorithm is run for every node s ∈ V. A node array "na" is used such that the entry na[v], v ∈ V, stores the first edge (s, x) in a shortest s-v path in G. This is constructed in a way similar to the shortest path tree: every time the distance label of a node v is adjusted via (u, v), we set na[v] to (u, v) if u=s and to na[u] otherwise (Lines 11 -14 of code segment 3).

```
1  while ( !PQ.empty() ) {
2    node u = PQ.del_min();
3    if(u != s) {
4      ea[ na[u] ].addPoint( ncoord[u] );
5    }
6    forall_out_edges(e,u) {
7      v = target(e);
8      double c = dist[u] + cost[e];
9      if( c < dist[v] ) {
10       PQ.decrease_p(v,c);  dist[v] = c;
11       if(u==s)
12         na[v]=e;
13       else
14         na[v]=na[u];
15     }
16   }
17 }
```

Code Segment 3. Container Construction

When a node u is removed from the priority queue PQ, na[u] holds the outgoing edge with which a shortest path from s to u starts. This information is stored in an edge array "ea". Line 4 invokes the container update routine for associating the vertex 'u' with the appropriate edge.

The problem that arises from using S (e) is the space requirements. Storing with each edge, a list of target nodes that can be reached using it would require O (mn) space where number of edges is m and the number of nodes is n; this is substantially large for a sparse graph.

Using geometric objects (geometric containers) the space required for storing preprocessed information can be reduced. The impact of using the containers to speedup Dijkstra's algorithm does depend on the relation of layout and edge weights. The containers are best suited for constant graphs because for dynamic graphs where the edge weights change rapidly results in updating the containers which is a costly operation and it requires more time. A container can have wrong nodes. These wrong nodes get naturally added up when the targets associated with a particular edge are far apart in the original layout of the graph.

2) Bounding Box**:** The geometric object used for testing this speedup technique is the bounding box[6] shown in Figure 1. It suffices to store four numbers for each object, which are the lower, upper, left and right boundary of the box. The bounding boxes can easily be computed online while the shortest paths are computed in the pre-processing.

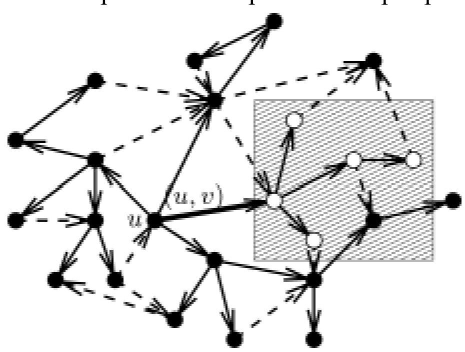


Figure 1. Bounding Box from[6]

Expansion of Bounding Box

The operations involved in computing shortest paths using geometric containers are creating consistent container, enlarging the container associated with each edge and then checking containment of a node within the bounding box while computing the shortest path. The steps involved in creating consistent containers are given in code segment 3. Enlarging the container for each edge to include the target nodes is performed in code segment 4. Each node is associated with a coordinate value obtained from the layout of the given graph. If a new vertex is to be added to the container associated with an edge, the co-ordinate values of the new vertex is compared with the existing boundary co-ordinates. The co-ordinates of the containers are adjusted if necessary to include the newly added vertex.

```
bool addPoint( const CPoint &p ) {
    if( p.x < min_x )
        min_x = p.x;
    else if( p.x > max_x )
        max_x = p.x;

    if( p.y < min_y )
        min_y = p.y;
    else if( p.y > max_y )
        max_y = p.y;

    return true; //Sucessflly updated the container
}
```

Code Segment 4. Expanding the Bounding Box

While computing the shortest path, when an edge e is reached, the boundary values of that edge e is checked to see if it contains the target node. If the target is present in the container then the edge is selected otherwise the edge is discarded. Code fragment 5 checks if a given node (specifically if the target) is present in a container.

```
bool contains(const CPoint &p) const {
    if( p.x >= min_x && p.x <= max_x &&
        p.y >= min_y && p.y <= max_y )
        return true;
    else
        return false;
}
```

Code Segment 5. Checking the Container

C. COMBINATION OF LANDMARKS AND GEOMETRIC CONTAINERS

```
1   for all nodes u belongs to V
       set dist(u) := infinity
2   initialize priority queue Q with source s and dist(s) := 0
3   while priority queue Q is not empty
4      get node u with smallest tentative distance dist(u) in Q
4a     if u = t return
5      for all neighbor nodes v of u
6         if t belongs to C(u, v)
7            set new-dist := dist(u) + w(u, v) + potential(u)
8            if new-dist < dist(v)
9               if dist(v) = infinity
10                  insert neighbor node v in Q with priority
new-dist
11              else
12                  set priority of neighbor node v in Q to new-
dist
13              set dist(v) := new-dist
```

Algorithm 2. Combination of Landmarks and Geometric Containers

The shortest path computation technique that combines both the landmarks and geometric containers is given in Algorithm 2. The changes made to the traditional Dijkstra's algorithm are in lines 4a, 6 and 7. Line 4a terminates the search procedure once the target is reached. Line 6 utilizes the containers for checking if an edge will eventually lead to the specified target. Line 7 in the algorithm uses potential values obtained from landmarks to modify vertex priority.

Line 6 assumes the existence of such a container for its functioning. It is remembered that the container associated with an edge, gives details pertaining to the targets that are reachable, with this edge included in their shortest path. Line 7 uses heuristic values to orient the search towards the target.The benefits of both the containers and landmarks are coupled as follows. The vertex ‘u’ to be visited next is deleted from the priority queue in line 4. The main modification occurs in the neighbour distance updation logic. Traditional Dijkstra’s algorithm considers all the neighbours ‘v’ of the selected vertex ‘u’; using containers only a subset of the neighbours ‘v’ to be visited are considered (line 6 of the algorithm), thereby reducing the search space. Then for the selected neighbours ‘v’ the distance to be updated includes an estimate of the distance from the vertex to the target (line 7 of the algorithm) and this helps to focus the search towards the target.

IV. Experimental Analysis

The different speedup techniques for Dijkstra’s algorithm were implemented in C++ with the help of LEDA library version 6.2 (Library of Efficient Data Types and Algorithms) [10]. The graph and priority queue data structures as well as other utilities such precise time measurement function provided by LEDA were used in the implementation. The code was compiled using Microsoft ® 32-bit C/C++ Compiler (version 15.00.30729.01) and the experiments were performed on an Intel Core2Duo machine (2.20 GHz) with 1 GB RAM running Windows 7 32-bit operating system.All the speedup techniques were coded as separate functions, for instance, the bidirectional search and traditional Dijkstra’s algorithm were kept as separate modules. The random and planar graph generators provided by LEDA were used for generating graphs on which the modules were tested. The number of vertices visited during the shortest path computation and runtime were measured and used as metrics for comparing the different speedup techniques. The time required for preprocessing and shortest path computation was accurately measured by using the functionality offered by LEDA.

A. *ANALYSIS OF LANDMARKS ON RANDOM GRAPHS*

The following remarks could be made based on the tabulated values. The preprocessing time steadily increases with the number of vertices. This is acceptable because the distance between a landmark and all the remaining vertices are computed during preprocessing. The running time of the modified search procedure with landmarks included is either nearly equal to or slightly higher than that of the traditional Dijkstra’s algorithm. The performance with landmarks is expected to improve on real world graphs. The number of vertices visited is reduced by using landmarks. A speedup of nearly 1.2 is achieved.

TABLE I. Comparison of traditional Dijkstra’s algorithm with Landmarks based on running time and vertices visited during shortest path computation on random graphs

Vertex Count	Edge Count	Preprocessing Time (s)	Runtime [with Landmarks] (s)	Vertices Visited [Landmarks]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]
10000	75500	0.332	0.0489	3698	0.0408	5365
11000	85250	0.41	0.0703	5215	0.0543	6295
12000	90600	0.477	0.0678	4626	0.0451	4739
13000	91000	0.518	0.081	5793	0.0536	6378
14000	107800	0.582	0.0759	4629	0.0621	5629
15000	112500	0.642	0.0899	5682	0.0715	7940
16000	120800	0.678	0.0984	6274	0.0739	7712
17000	127500	0.765	0.129	7747	0.0899	9177
18000	133200	0.886	0.107	5814	0.105	10380
19000	142500	0.878	0.108	6139	0.0905	8535
20000	161000	0.969	0.149	8147	0.132	12390
21000	155400	1.03	0.152	8313	0.125	11826
22000	167200	1.06	0.194	11341	0.124	10558
23000	154100	1.04	0.189	11749	0.104	10420
24000	166800	1.07	0.168	10432	0.119	11639
25000	186250	1.27	0.187	10245	0.137	11857

Figure 1. shows the number of vertices visited by the search procedure with landmarks and that of traditional Dijkstra plotted against the number of vertices present in the graph. The number of vertices visited by searching with landmarks is considerably less in most searches.

B. *ANALYSIS OF LANDMARKS ON PLANAR GRAPHS*

The effect of using landmarks during shortest path comp-

Figure 1. Vertices visited during shortest path computation by traditional Dijkstra and search procedure with Landmarks on random graphs

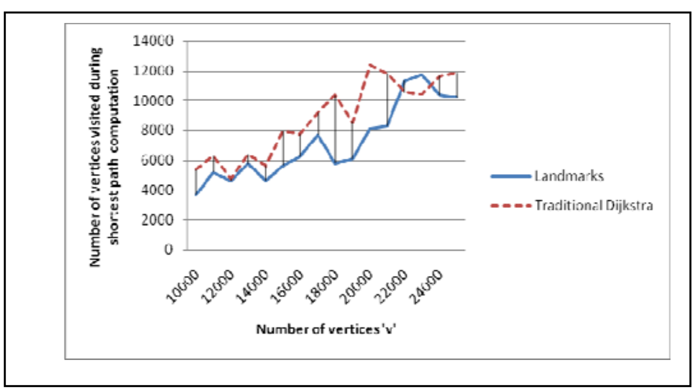
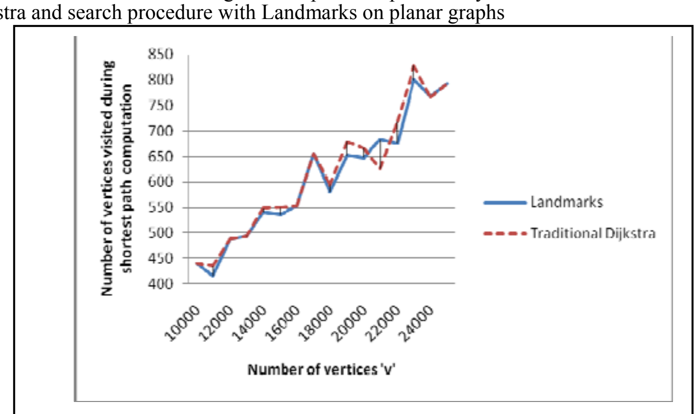


Figure 2. Vertices visited during shortest path computation by traditional Dijkstra and search procedure with Landmarks on planar graphs



-utation on planar graphs is analysed below. The performance of the technique in this graph type is nearly equal to that of traditional Dijkstra's algorithm. Figure 2 plots the vertices visited by traditional Dijkstra and a search using landmarks on the planar graphs generated by LEDA. The values are tabulated in Table 2.

Table II Comparison of traditional Dijkstra's algorithm with Landmarks based on running time and vertices visited during shortest path computation on planar graphs

Vertex Count	Edge Count	Preprocessing Time (s)	Runtime [with Landmarks] (s)	Vertices Visited [Landmarks]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]
10000	17509	0.0193	0.00475	439	0.0044	439
11000	19344	0.0208	0.00505	416	0.00445	434
12000	21166	0.0226	0.00555	488	0.00515	488
13000	22999	0.0248	0.0058	494	0.0053	493
14000	24869	0.0269	0.0063	540	0.00565	548
15000	26698	0.0282	0.0068	536	0.00635	550
16000	28555	0.0304	0.00725	554	0.00655	554
17000	30399	0.0358	0.00945	656	0.0087	656
18000	32251	0.0383	0.00955	580	0.00905	594
19000	34111	0.0412	0.01	652	0.0093	679
20000	35966	0.0452	0.0112	648	0.0102	666
21000	37844	0.0479	0.0113	683	0.0109	626
22000	39705	0.0518	0.0127	675	0.0112	718
23000	41550	0.0578	0.0139	801	0.0146	828
24000	43435	0.0603	0.0144	767	0.0129	766
25000	45286	0.0655	0.0144	794	0.0136	794

C. *PERFORMANCE OF GEOMETRIC CONTAINERS ON RANDOM GRAPHS*

Table 3. shows the experimental values obtained by comparing Geometric Containers with the traditional shortest path computation technique. A speedup of 1.2 is achieved based on the number of vertices visited during the shortest path computation. The average running time of the search with geometric containers nearly equals that of traditional search. Two important points of interest are as follows. The first one is that increasing the number of vertices can reduce the running time but due the memory limitations of the experimental setup and the libraries used the vertex count was not increased during the analysis. The second point to note is that geometric containers have a better performance in real word graphs and this was not tested due to time limitations.

Table III. Runtime and Number of vertices visited comparison of Geometric containers and traditional Dijkstra on random graphs

Vertex Count	Edge Count	Pre-processing Time (s)	Runtime [Containers] (s)	Vertices Visited [Containers]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]
1000	8000	1.879	0.0013	512	0.001	325
2000	16000	7.99	0.0026	984	0.0028	1204
3000	21000	17.9	0.0042	1430	0.0038	1592
4000	20000	27.9	0.0037	1316	0.0048	2306
5000	25000	47.1	0.006	1778	0.006	2749
6000	30001	73.5	0.0086	2749	0.007	3000
7000	63000	148	0.0138	3233	0.0108	2959
8000	72000	214	0.0128	2640	0.0139	3596
9000	72000	257	0.0158	3224	0.0182	5373
10000	100000	389	0.029	5634	0.0229	5312

Figure 5.Vertices visited during shortest path computation by traditional Dijkstra and search procedure with containers on random graphs

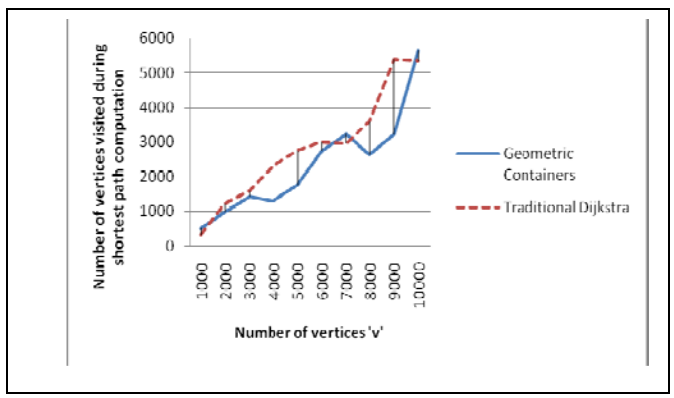


Figure 5 shows the number of vertices visited by traditional search technique and the search with containers, plotted against the number of vertices present in the graph.

D. *PERFORMANCE OF GEOMETRIC CONTAINERS ON PLANAR GRAPHS*

Using geometric containers on planar graphs generated by LEDA had a meagre performance on the number of vertices visited. The values are shown in Table 4. It gives a varying results of running time and vertices visited compared to that of Dijkstra.

Table IV. Runtime and visited vertices comparison of Geometric containers and traditional Dijkstra on planar graphs

Vertex Count	Edge Count	Preprocessing Time (s)	Runtime [with Containers] (s)	Vertices Visited [Containers]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]
10000	17470	25.085	0.0031	116	0.0047	106
11000	19294	31.3	0.0015	78	0.0063	122
12000	21102	36.9	0.0045	76	0.0048	103
13000	23010	43.6	0.0047	99	0.0047	123
14000	24857	50.5	0.0045	82	0.0048	156
15000	26756	58.3	0.0063	119	0.0046	60
16000	28556	66.2	0.0046	164	0.0063	46
17000	30407	78	0.0125	160	0.0031	108
18000	32180	87.6	0.014	118	0.0016	98
19000	34164	98.1	0	112	0.0156	162
20000	35953	109	0.0156	59	0	114

E. COMBINATION OF LANDMARKS AND GEOMETRIC CONTAINERS APPLIED TO RANDOM GRAPHS

The important technique implemented in this work is a search procedure that combines both the landmarks and geometric containers during shortest path computation. The graphs considered for analysis by this technique are undirected, though it is possible to apply the technique to directed-graphs, minor changes to the landmarks module will be necessary. Table V. shows the experimental results. Though the preprocessing time is high, the process occurs only once and therefore excluded from the shortest path computation runtime values. The new technique reduces the number of vertices visited during the shortest path query evaluation and speedup of 1.79 based on vertex-visit-count is achieved. The values in Table V. are with respect to random graphs generated by LEDA.

There is huge difference in pre-processing time of landmarks and containers. As the container construction itself will take a longer time, combination of landmark and container will have more time. The difference steadily increases with the number of nodes.

If running time alone is considered the traditional algorithms work better in some case, but large sparse graph it is always necessary to consider the preprocessing time also. Here, additional advantage is for the number of nodes visited during the search. i.e the speedup is measured in terms of number of nodes visited. The speedup is expected to improve for the increasing number of nodes and for real world graphs.

F. *Combination of Landmarks and Geometric Containers applied to Planar Graphs*

A speedup of 1.12 was achieved based on the running time of the technique whereas the speedup was 1.7 with respect the number of vertices visited during the shortest path computation. Eventhough the preprocessing phase occurs only once, the preprocessing time is considerably reduced in planar graphs for containers. The number of vertices visited is reduced compared to that of Dijkstra's algorithm.

Figure 6. compares the running time of the combined speedup technique with that of Dijkstra's and the average running time is observed to be slightly improved. As mentioned earlier containers perform well when applied to real world graphs and hence the combined speedup technique is also expected to perform better in such a scenario.

Figure 7. is the graph, which gives the variations of the number of vertices visited by the techniques under comparison, viz., "combined landmarks and containers" and traditional Dijkstra's algorithm.

From the tabulated values it can be inferred that the combined speedup technique improves the performance of shortest path computation to a considerable extent.

Table V. Combination of Landmarks and Geometric Containers compared with traditional Dijkstra on Random Graphs

Vertex Count	Edge Count	Preprocess Time [Landmarks]	Runtime [Combination] (s)	Vertices Visited [Combination]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]	Preprocess Time [Containers]
1000	10000	0.019	0.001	85	0.0015	276	3.13
2000	20000	0.0325	0.0055	552	0.003	348	13.8
3000	30000	0.0725	0.007	579	0.011	2597	38.1
4000	40000	0.099	0.022	1801	0.014	2699	80.3
5000	50000	0.139	0.027	1993	0.0195	3040	145
6000	60000	0.227	0.0155	892	0.013	1495	229
7000	70000	0.237	0.0325	1884	0.038	5501	339
8000	80000	0.348	0.0244	1228	0.0355	4868	467
9000	90000	0.347	0.0405	2760	0.0375	3067	633
10000	100000	0.296	0.0386	2062	0.0080	944	781

Table VI. Combination of Landmarks and Geometric Containers compared with traditional Dijkstra on Planar Graphs

Vertex Count	Edge Count	Preprocess Time [Landmarks]	Runtime [Combination] (s)	Vertices Visited [Combination]	Runtime [Dijkstra] (s)	Vertices Visited [Dijkstra]	Preprocess Time [Containers]
1000	1519	0.0025	0	40	0.001	124	0.256
2000	3213	0.0035	0.0005	71	0.001	123	1
3000	4932	0.005	0.000999	119	0.001	184	2.26
4000	6659	0.007	0.001	69	0.002	226	4.01
5000	8435	0.009	0.002	74	0.002	199	6.26
6000	10249	0.011	0.002	133	0.003	298	9.05
7000	12075	0.012	0.003	361	0.003	445	12.4
8000	13878	0.013	0.0035	310	0.003	437	16.3
9000	15696	0.017	0.0035	234	0.004	349	20.7
10000	17512	0.0185	0.005	356	0.004	650	25.6

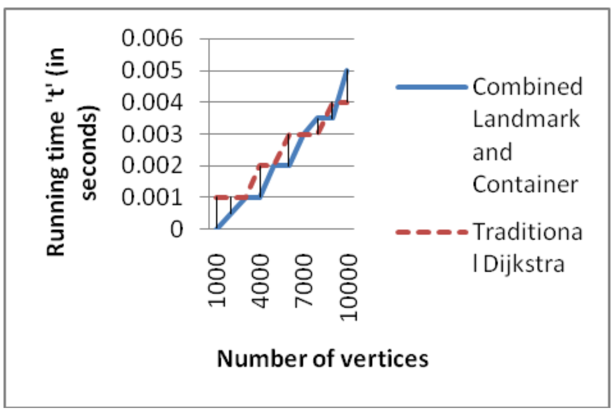


Figure 6. Running time of the "combined landmark and container" speedup technique compared with Dijkstra's algorithm

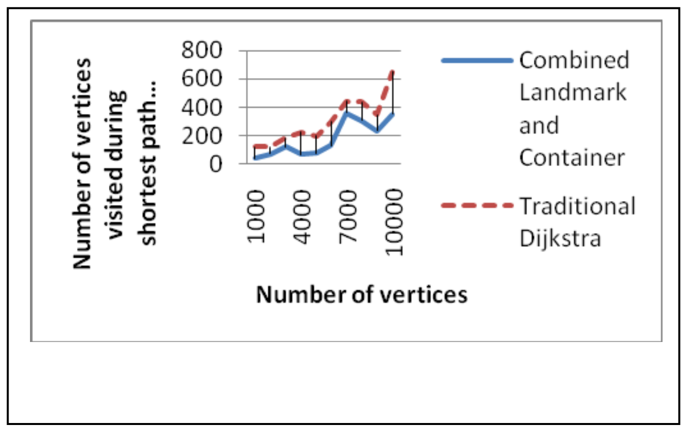


Figure 7. Vertices visited using the "combined landmark and container" speedup technique compared with Dijkstra's algorithm

V. CONCLUSION

The speedup techniques used for Dijkstra's algorithm like Landmarks and Geometric containers were analysed with random graphs and planar graphs. The key metrics for evaluation of the techniques like speedup based on running time and the number of vertices visited during shortest path computation were considered. The technique of combining landmarks and geometric containers was also analysed for the same random and planar graph types.

Each speedup technique worked well for a specific type of graph and hence the performance was appreciable in those cases. The heuristic values obtained by using landmarks helped to reduce the number of vertices visited during shortest path computation but the running time of the technique was marginally high due the computation overhead involved during vertex distance updation process. The geometric containers achieved speedup based on the vertices visited during the evaluation of shortest path query but were nearly equal in running time to the traditional search process. The combined speed up technique based on landmarks and containers was able to perform better under the same experimental setup compared to the other techniques. Based on the running time the speedup was 1.12 while based on the number of vertices visited the speedup attained was 1.7.

The performance is expected to be improved on real world graphs compared to the graphs generated by LEDA. The technique can be extended for new combinations. This technique can also be applied to various other graph types.

ACKNOWLEDGMENT

The authors wish to thank Mr. R. Arvind Kumar for his efforts in conducting the experiments.

REFERENCES

[1] DIJKSTRA, E. W, "A note on two problems in connection with graphs", In Numerische Mathematik, Vol. 1, Mathematisch Centrum, Amsterdam, The Netherlands, 1959, pp269–271.

[2] Andrew V.Goldberg & Chris Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory", In Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms, 2005.

[3] I. Phol, "Bi-directional Search", In Machine Intelligence, volume 6, 1971, pp 124-140. Edinburgh Univ. Press, Edinburgh.

[4] Martin Holzer, "Hierarchical speedup techniques for shortest path algorithms", M, Tech. report, Dept of Informatics, 2003, vUniversity of Konstanz, Germany.

[5] Frank Schulz, Dorothea Wagner, & Christos Zaroliagis, "Using multi-level graphs for timetable information in railway systems", In Proc. 4th Workshop on Algorithm Engineering and Experiments. LNCS 2409, Springer-Verlag, New York. , 2002, pp43- 59.

[6] Dorothea Wagner, Thomas Willhalm, & Christos Zaroliagis, "Geometric Containers for Efficient Shortest-Path Computation", ACM Journal of Experimental Algorithmics, 10(1.3), 2005.

[7] Martin Holzer, Frank Schulz, Dorothea Wagner, & Thomas Willhalm, "Combining Speed-up Techniques for Shortest-Path Computations", *ACM Journal of Experimental Algorithmics*, Vol. 10, Article No. 2.5., 2005.

[8] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, & Dorothea Wagner, " Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm", ACM Journal of Experimental Algorithmics, Vol. 15, Article No. 3, 2010.

[9] Bauer. R, Delling. D, Sanders. P, Schieferdecker. D, Schultes. D & Wagner. D, "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm", in the proceedings of the 7th Workshop on Experimental Algorithms(WEA'08), Springer, Berlin, pp.303-318, 2008.

[10] LEDA, http://www.algorithmic-solutions.com

AUTHORS PROFILE

R. Kalpana is working as Associate Professor in the Department of Computer Science & Engineering, Pondicherry Engineering College, Pondicherry, India.She has completed her B.Tech(Computer Science & Engineering) from Pondicherry University.She has completed her M.Tech(Computer Science & Engineering) from Pondicherry University. She is pursuing Ph.D at Pondicherry University. She has published ten papers in national and international conferences/ journals.

Prof. Dr. P. Thambidurai is the Principal and Professor of Computer Science & Engineering, Perunthalaivar Kamarajar College of Engineering and Technology, Karaikal. He has completed his M.E(Computer Science & Engineering) from Anna University, Chennai. He has completed his Ph.D from Alagappa University, Karaikudi, Tamilnadu. He has published more than hundred papers in national, International journals and conferences.