KaLP v1.00 – Karlsruhe Longest Paths
User Guide

Tomas Balyo, Kai Fieger and Christian Schulz
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Email: {balyo, christian.schulz}@kit.edu, fieger@ira.uka.de

Abstract

This paper severs as a user guide to the longest paths framework KaLP (Karlsruhe Longest Paths). We give a rough overview of the techniques used within the framework and describe the user interface as well as the file formats used.

Contents

1 Introduction

The longest path problem (LP) is to find a simple path of maximum length between two given vertices of a graph where length is defined as the number of edges or the total weight of the edges in the path. The problem is known to be NP-complete [2] and has several applications such as designing circuit boards [6, 5], project planning [1], information retrieval [10] or patrolling algorithms for multiple robots in graphs [7]. For example, when designing circuit boards where the length difference between wires has to be kept small [6, 5], the longest path problem manifests itself when the length of shorter wires is supposed to be increased. Another example application is project planning/scheduling where the problem can be used to determine the least amount of time that a project could be completed in [1].

The purpose of the manual is to give a very rough overview over the techniques used in the programs, as well as to serve as a guide and manual on how to use our algorithms. We start with a short overview of the algorithms implemented within our framework. This is followed by a description of the graph format that is used. It is basically the same graph format that is used by Metis [4] and Chaco [3], as well as the DIMACS graph format. We then give an overview over the user interface of KaLP.

2 Longest Paths by Dynamic Programming

We now give a rough overview over the algorithms implemented in our framework. For details on the algorithms and further improvements, we refer the interested reader to the corresponding papers. Our algorithm solves the longest path problem (LP) for weighted undirected graphs. We restrict ourselves here to introducing the main approach which includes preprocessing the graph as well as combining paths.

2.1 The Basic Approach

A simple way to solve the longest path problem is *exhaustive depth-first search* [9]. A regular depth-first search (DFS) starts at a root vertex. By default, a vertex has two states: marked and unmarked. Initially, all vertices are unmarked, except the root which is marked. DFS calls itself recursively on each unmarked vertex reachable by an edge from the current vertex, which is the parent of these vertices. The vertex is marked. Once it is done it backtracks to its parent. The search is finished once DFS backtracks at the root vertex.

Exhaustive DFS is a DFS that unmarks a vertex upon backtracking. In that way every simple path in the graph starting from the root vertex is explored. The LP problem can be solved with exhaustive DFS by using the start vertex as root. During the search the length of the current path is stored and compared to the previous best solution each time the target vertex is reached. If the current length is greater than that of the best solution, it is updated accordingly. When the search is done a path with maximum length from s to t has been found. The *main idea* of LPDP is to partition a graph into multiple blocks and run a search similar to exhaustive DFS on each block in a preprocessing step. Afterwards the results can be combined into a single longest path for G.

2.2 Partitioning and Preprocessing

We now explain our preprocessing routine. First of all, we partition a graph into a predefined number of blocks and modify the input instance. Partitioning can be done using a graph partitioning algorithm, e.g. KaHIP [8]. We then *replace* every cut edge $e = \{x, y\}$, i.e. an edge running between two blocks, by introducing two new vertices v_e,v'_e and the edges $\{x, v_e\}$ and $\{v'_e, y\}$. One of these edges retains the weight of the original edge e, the other edge weight is set to zero. Additionally, we insert two new vertices. One is connected to the start vertex and the other one to the target vertex. In both cases, we use an edge having weight zero. All newly generated vertices are called *auxiliary-vertices* throughout the rest of the section. Next, we compute subgraphs $G_A := (V_A, E_A)$ where V_A is the set vertices of block A as well as all auxiliary-vertices connected to them, and E_A are all edges that run between those vertices. See Figure 2 for an example.

Now observe the following property: for a longest simple path in G the auxiliary-vertices function as entry and exit points for their block of the partition. That means a longest simple path from s to t can only enter and exit a block through the inserted auxiliary-vertices. For a block that does not contain s or t, every time the path enters a block, it also has to leave it again, connecting auxiliary-vertices in pairs of two. The sets of auxiliary-vertex-pairs $\mathcal{P}_A$ that we are interested in for any subset s of the auxiliary-vertex-pairs for any block are equivalent to the matchings that exist for a clique-graph consisting of all auxiliary-vertices of that block. In other words, each endpoint can only appear once in a subset of pairs. The pairs have to be connected by *non-intersecting* simple paths. Our preprocessing algorithm computes the longest of these connections for each block and set of auxiliary-vertex-pairs with a modified version of exhaustive DFS executed on the respective subgraph, whose descriptions follows.

Our modified exhaustive DFS is executed multiple times each time using a different auxiliary-vertex as a root. The search algorithm divides its current search path into different path segments. It traverses the vertices of the graph as usual with the exception of the auxiliary-vertices. The first segment starts from the root auxiliary-vertex. The segment is completed once a different auxiliary-vertex is reached. When this happens, the algorithm starts a new segment by jumping to an other auxiliary-vertex and continues traversing the graph as before. This way each segment starts and ends in a auxiliary-vertex. The start- and endpoints of all segments resemble the auxiliary-vertex

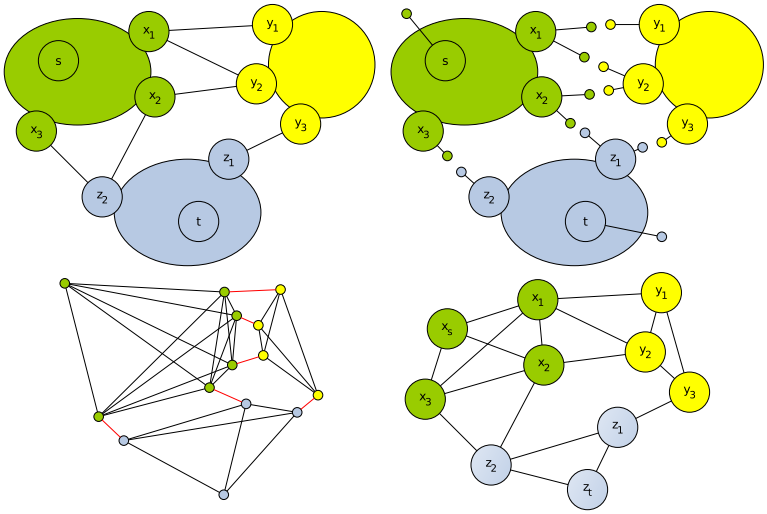


Figure 1: An example illustrating the basic idea of the LPDP algorithm. In the upper left corner we have a graph partitioned into three blocks indicated by the colors green, yellow and blue. The starting node is in the green block and the target in the blue block. The border nodes are named x_i, y_i and z_i. In the upper right corner we see the three graphs created by removing the edges connecting the blocks and adding the auxiliary nodes (indicated as small circles). Finally, in the bottom we have the graphs used in the second stage of the algorithm – combining paths. On the left side we see the simple version and right side contains the improved version with fever auxiliary nodes and vertices.

pairs $\mathcal{P}_A$. The best current result for each possible element in $\mathcal{P}_A$ is stored and updated if necessary every time a path segment is completed. To avoid unnecessary traversal (due to symmetry) of the graph, a path segment is only allowed to end in a vertex having an id higher than its start vertex. Additionally a path segment can only start from an auxiliary-vertex, if this vertex is higher than all other starting vertices in the current search path.

2.3 Combining Paths

After having performed preprocessing, we can now find the longest (simple) path of G in an auxiliary graph that only contains auxiliary-vertices and edges between them. In this graph two auxiliary-vertices are connected by an edge if they belong to the same graph G_A. Note that every block of the original partition is now represented by a clique of its auxiliary-vertices in our auxiliary graph. Additionally, the edges $\{v_e, v'_e\}$ get introduced for all edges e that were replaced in the previous graph, where v_e and v'_e are the auxiliary-vertices that replaced e. These edges represent the connections between the blocks.

In order to solve the longest path problem, we use another modified version of exhaustive DFS. It starts from the vertex representing the start vertex. The algorithm creates a set of auxiliary-vertex-pairs $\mathcal{B}_A \in \mathcal{P}_A$ for every block A from its search path. An edge $\{v, w\}$ is part of a block A if both v and w are auxiliary-vertices of A. While this edge is part of the search path the pair $\{v, w\}$ is an element of $\mathcal{B}_A$. The pair $\{v, w\} \in \mathcal{B}_A$ represents a connection of the corresponding auxiliary-vertices of v and w in G_A through a simple path. The simple paths of all these pairs in $\mathcal{B}_A$ cannot intersect with each other. The best possibility to do this and the combined length of these paths has already been calculated in the preprocessing phase. We do the following in order to only receive valid auxiliary-vertex pairs $\mathcal{B}_A$ when trying to append new edges to the current search path: first of all, for every block A a solution for $\mathcal{B}_A$ has to exist. This can be looked up, since the best possible solutions have been calculated in the preprocessing step. Second, we have to search the graph in an alternating pattern between edges that are part of a block and edges that connect two blocks. Otherwise $\{a, b\}, \{b, c\} \in \mathcal{B}_A$ would be possible, which means

that the two paths in G_A would intersect. Now every time the vertex representing of the original target-vertex has been found, the paths in G_A for every $\mathcal{B}_A$ are looked up and their weight summed up. At the end the different $\mathcal{B}_A$ with the highest combined weight are returned. This weight is the weight/length of the longest simple path in G. The actual longest simple path can be constructed by looking up all the pre-calculated paths in G_A for the given connections of its auxiliary-vertices $\mathcal{B}_A$.

3 Graph Format

3.1 Input File Format

We use two file formats: the Metis/Chaco format and the DIMACS graph format.

Metis/Chaco Graph Format. The first graph format used by our programs is the same as used by Metis [4], Chaco [3] and the graph format that has been used during the 10th DIMACS Implementation Challenge on Graph Clustering and Partitioning. If you want to use this format, your filename has to end with the ".graph" extension. The input graph has to be undirected, withoud self-loops and without parallel edges.

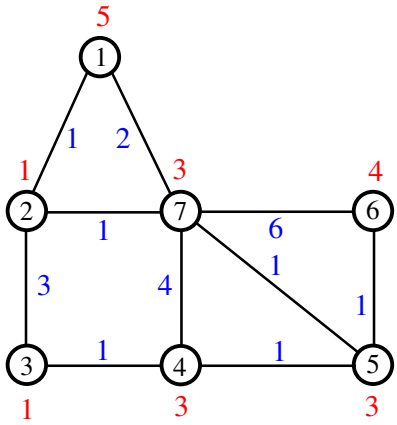
To give a description of the graph format, we follow the description of the Metis 4.0 user guide very closely. A graph $G = (V, E)$ with n vertices and m edges is stored in a plain text file that contains $n + 1$ lines (excluding comment lines). The first line contains information about the size and the type of the graph, while the remaining n lines contain information for each vertex of G. Any line that starts with % is a comment line and is skipped.

The first line in the file contains either two integers, n m, or three integers, n m f. The first two integers are the number of vertices n and the number of undirected edges of the graph, respectively. Note that in determining the number of edges m, an edge between any pair of vertices v and u is counted *only once* and not twice, i.e. we do not count the edge (v, u) from (u, v) separately. The third integer f is used to specify whether or not the graph has weights associated with its vertices, its edges or both. If the graph is unweighted then this parameter can be omitted. It should be set to 1 if the graph has edge weights, 10 if the graph has node weights and 11 if the graph has edge and node weights.

The remaining n lines of the file store information about the actual structure of the graph. In particular, the ith line (again excluding comment lines) contains information about the ith vertex. Depending on the value of f, the information stored in each line is somewhat different. In the most general form (when $f = 11$, i.e. we have node and edge weights) each line has the following structure:

$$c\, v_1\, w_1\, v_2\, w_2 \ldots v_k\, w_k$$

where c is the vertex weight associated with this vertex, $v_1, \ldots, v_k$ are the vertices adjacent to this vertex, and $w_1, \ldots, w_k$ are the weights of the edges. Note that the vertices are numbered starting from 1 (not from 0). Furthermore, the vertex-weights and edge-weights must be integers greater or equal to 0. However, note that vertex-weights are ignored by our algorithm.




```
7  10 11
5  2  1  7  2
1  1  1  3  3 7 1
1  2  3  4  1
3  3  1  7  4  5 1
3  4  1  7  1  6 1
4  5  1  7  6
3  1  2  2  1  4  4  5  1  6  6
```

Figure 2: An example graph and its representation in the Metis/Chaco graph format. The IDs of the vertices are drawn within the cycle, the vertex weight is shown next to the circle (red) and the edge weight is plotted next to the edge (blue).

DIMACS Graph Format. The second graph format that our software can read is the DIMACS graph format from the 9th DIMACS Implementation challenge. Here, a graph contains n nodes and m arcs. Nodes are identified by integers $1 \ldots n$. Graphs in the format can be interpreted as directed or undirected, depending on the problem being studied. However, for our software to work correctly, there needs to be a backward edge for each directed edge, i.e. the graph needs to be undirected. Arc weights are signed integers. By convention, graph file names should have the suffix ".gr" or ".dimacs". Line types are as follows:

- comment lines can appear anywhere and are ignored by programs. Example: "c This is a comment".
- the problem line is unique and must appear as the first non-comment line. Example "p sp n m". Here, n is the number of nodes and m is the number of arcs.
- arc descriptor lines describe arcs and their weights. Example: "a u v w". Here, u is the source vertex, t is the target vertex and w is the weight/length associated with the arc.

An example file for the graph from Figure 2:

p sp 10 22
c this is a comment
c the graph contains 10 nodes and 22 arcs
c node ids are numberd in 1..10
a 1 2 1
a 1 7 2
a 2 1 1
a 2 3 3
a 2 7 1
a 3 2 3
a 3 4 1
a 4 3 1
a 4 7 4
a 4 5 1
....

3.2 Output File Formats

3.2.1 Path File

We now specify our output format. This file contains $\ell + 1$ lines where ℓ is the number of edges in the longest path. Let $p := v_1, v_2, \ldots, v_\ell$ be the longest path from $s = v_1$ to $t = v_\ell$. In each line the ID of the corresponding vertex is given, i.e. line i contains the ID of the vertex v_i (here the vertices are numbered from 0 to $n - 1$).

3.3 Troubleshooting

KaLP should not crash! If KaLP crashes it is mostly due to the following reasons: the provided graph contains self-loops or parallel edges, there exists a forward edge but the backward edge is missing or the forward and backward edges have different weights, or the number of vertices or edges specified does not match the number of vertices or edges provided in the file. In case of the METIS format, please use the *graphchecker* tool provided in our package to verify whether your graph has the right input format. If our graphchecker tool tells you that the graph that you provided has the correct format and KaLP crashes anyway, please write us an email.

4 User Interface

KaLP contains the following programs: kalp, graphchecker, generate_maze. To compile these programs you need to have Argtable, g++ and scons installed (we use argtable-2.10, g++-4.8.0 and scons-1.2). Once you have that you can execute *compile.sh* in the main folder of the release. When the process is finished the binaries can be found in the folder *deploy*. We now explain the parameters of each of the programs briefly.

4.1 KaLP

Description: This is the longest path program.

Usage:

kalp file --start_vertex=<int> --target_vertex=<int> [--help]
[--print_path] [--partition_configration=<string>] [--output_filename=<string>]

Options:

file	Path to graph file that you want to partition.
--help	Print help.
--start_vertex=<int>	Start vertex to use.
--target_vertex=<int>	Target vertex to use.
--print_path	Printing the solution at the end of the program.
--partition_configuration=<string>	Use a configuration for the partitioning tool. (Default: eco) [strong|eco|fast]. We recommend to use the strong configuration on difficult instances.
--output_filename=<string>	Output filename. If specified the vertices of the longest path will be written into that file.

4.2 Generate Maze

Description: This is a program that generates a maze, from that generates a graph and writes the graph to disc using the DIMACS format..

Usage:

generate_maze [--help] [--output_filename=<string>] [--seed=<int>]
[--width=<int>] [--height=<int>] [--blocked=<double>] [--print_path]

Options:

--help	Print help.
--output_filename=<string>	Filename of the outputfile. Default: grid.dimacs
--seed=<int>	Seed to use for random number generator.
--width=<int>	Width of the maze. Default: 10
--height=<int>	Height of the maze. Default: 10
--blocked=<double>	Percentage of cells in the maze to be blocked. Default: 0.3 (i.e. 30%)
--print_path	Printing the solution at the end of the program.

4.3 Graph Format Checker

Description: This program checks if the graph specified in a given file is valid (only for METIS file format).

Usage:
graphchecker file

Options:

file Path to the graph file.

References

[1] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[3] B. Hendrickson. Chaco: Software for Partitioning Graphs. `http://www.cs.sandia.gov/~bahendr/chaco.html`.

[4] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[5] M. M. Ozdal and M. D. F. Wong. A Length-Matching Routing Algorithm for High-Performance Printed Circuit Boards. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2784–2794, 2006.

[6] M. M. Ozdal and M. D. F. Wong. Algorithmic study of single-layer bus routing for high-speed boards. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):490–503, 2006.

[7] D. Portugal and R. Rocha. MSP Algorithm: Multi-robot Patrolling Based on Territory Allocation Using Balanced Graph Partitioning. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1271–1276, New York, USA, 2010. ACM.

[8] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

[9] R. Stern, S. Kiesel, R. Puzis, A. Feller, and W. Ruml. Max is more than min: Solving maximization problems with heuristic search. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 2014.

[10] W. Y. Wong, T. P. Lau, and I. King. Information Retrieval in P2P Networks Using Genetic Algorithm. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 922–923, New York, USA, 2005. ACM.