A HEURISTIC TO GENERATE ALL BEST PARTIALLY DISJOINT PATHS IN A COMMUNICATIONS NETWORK

Alexander A. Kist, Richard J. Harris

RMIT University Melbourne, BOX 2476V, VIC 3001, Australia, {kist,richard}@catt.rmit.edu.au

Abstract - This paper considers a directed graph $G(N, A)$ with n nodes, m directed arcs and a cost value for each arc. The synthesis of a partially link disjoint pair of paths for a given OD-pair with the minimum total cost is investigated. An algorithm is presented that solves the all-best partially disjoint path problem. The expected worst case running time of the algorithm is $O(n^3)$. Possible applications include light path design and MPLS based traffic engineering. Partially disjoint path algorithms can also be used as intelligent alternatives to k-th shortest path algorithms.

Keywords - Partially disjoint paths, shortest path, virtual path synthesis, backup path creation.

I. INTRODUCTION

Modern communication networks widely use backup paths to ensure the availability of resources in the case of link failure. Feasible backup paths have to be *link disjoint* to provide the best protection. This paper discusses the problem of finding two partially disjoint paths with a minimum cost in a communication network. Parts of the paths between an OD-pair use the same hops and parts of the paths are completely link-disjoint for a partially disjoint paths pair.

Reasons for requiring such shortest path pairs are: Firstly, under certain conditions it is not necessary to generate complete link disjoint paths to meet the preconditions. The weakening of the complete disjointness only makes sense when the partially disjoint path pair has a better global cost than the original shortest complete disjoint path pair. Secondly, in parts of the network, no two link disjoint paths are available. In such a case, it is necessary to find the best partially disjoint path pair.

Where backup path generation and load balancing have been extensively investigated in the past e.g. [1] and [2], the problem of partial link disjointness has not being considered. Iwata [3] discusses a special case of partially link disjointness where the shared paths are already known. Brander gives in [4] a comprehensive study between several n-th shortest path algorithms without the disjointness condition. The motivation for this work came from Saayan Choudhury and his PhD thesis [5] where he presents a planning model for virtual path design in ATM networks. Furthermore this problem occurs in relation to all virtual path connections. Possible applications include light path design and MPLS based traffic engineering. Partially disjoint path algorithms can also be used as intelligent alternatives to k-th shortest path algorithms.

The algorithm, introduced in this paper, is capable of finding all best partially disjoint paths between an origin-destination (OD) pair. The partially disjoint path problem is discussed in [6] in detail. Different network topologies are investigated and the degree of divergence is defined as a value to judge the independence of paths. The observation in [6] of different topologies for the complete and partially link-disjoint case yields that the commonly used paths have to be equal to paths of the original single shortest path to provide feasible shared hops. The algorithmic approach in this paper is based on the Suurballe algorithm ([7], [8]) and uses an efficient Dijkstra implementation [9] for the shortest path algorithms. The partially disjoint paths are collected in a specially designed data structure during the execution of the modified algorithm. It is called *partially disjoint path label* (PDPL).

Section II introduces the PDPL and discusses the operations on this label. Section III describes formally the steps of the network algorithm. Section IV discusses the correctness and performance of the algorithm.

II. PARTIALLY DISJOINT PATH LABEL

The algorithm requires a node label that is capable of storing the different partially disjoint paths. The *partially disjoint path label* (PDPL) is a sorted list with k items. Every item in the list specifies a path and its cost attributes. The three elements are: A distance label c_i to the origin node. Label $c(i)$ is known from several algorithms (eg. Dijkstra [10]). The cost c_i is the cost from the O node on the indicated path to the actual node. A cost label $u(i)$ that consists of the cost of the components which are used by both paths. The used cost u is the accumulated cost of arcs that are part of the shortest path and the path indicated in $path$. And the actual path is stored in the $path$ variable. The costs c and u apply to this path. $path_i$ is represented as a sequence of nodes stored in a list.

The items in the list are sorted and only the best paths are stored. Figure 1 shows example labels. The degree of the label k is equal to the number of entries in the field. Every item in the label-list represents a partially disjoint path. The cost label u_i is strongly increasing from the top item to the bottom item, the cost label c_i is strongly decreasing from the top item to the bottom item. The label has to be maintained in a manner that ensures the properties of the list. The properties

are summarised by Equations (1).

$$\begin{array}{llll} \text{(i)} & u_k \leq U_{allowed} & \text{(ii)} & u_1 \geq 0 \\ \text{(iii)} & c_1 \leq C^{'}_{DPsingle} & \text{(iv)} & c_k \geq C_{SP} \end{array} \quad (1)$$

The highest common cost u_k is located in position k, the bottom item of the list. The upper bound for this cost is given by the allowed cost $U_{allowed}$. The lowest value for the common cost is zero. The complete disjoint path is stored in the first item. It has a common cost $u_1 = 0$ and a cost $C^{'}_{DPsingle}$, which is the upper cost bound. $C^{'}_{DPsingle}$ is the reduced cost of the complete disjoint path in the transformed graph. All items have to have a smaller cost value c_i. The smallest cost value c_i is located at the bottom item. It is the cost of the shortest path.

The motivation for this maintenance of the label is that only the best paths are stored. E.g. a path with a cost of 17 and used common path cost worth 2 is "better" than a path with the same cost and a higher value of common path cost, for example the pair $17 - 3$. An additional stored path must therefore have a lower cost value c for a higher value of u. Otherwise, the path provides no achievement. If an item after the insertion is in violation of the property of the $PDPL$, it is deleted. The label update is explained in the next section.

A. Label Update

Figure 1 shows a hop ab with the cost c_{ab} connecting two nodes a and b with assigned PDPLs. Label a has already valid entries. For the up-date of label b exist four possible cases: Label b can be empty ($k = 0$) or it has already a number of k items. The arc ab, connecting both nodes, is part of the SP or it is not part of the SP.

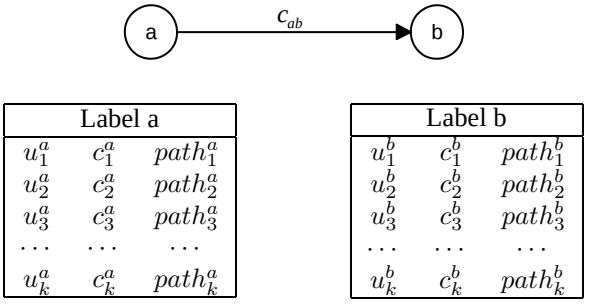

Fig. 1. PDPL - Update

(i) The degree of label b is **zero** and the arc ab is **not part** of the SP

$$\begin{array}{rcll} \forall i | i \leq k_a : \quad u_i^b & = & u_i^a & c_i^b = c_i^a + c_{ab} \\ path_i^b & = & path_i^a \cup \{a\} & \end{array} \quad (2)$$

Since ab is not part of the SP, the cost $u(i)$ is not changed. All arcs, stored in label a, are simply copied to label b with the added arc cost c_{ab}. After insertion the degree of label b is equal to the degree of label a.

(ii) The degree of label b is **zero** and the arc ab is **part** of the SP $\forall i | i \leq k_a \wedge u_i^a + c_{ab} \leq C_{allowed}$:

$$\begin{array}{rcll} u_i^b & = & u_i^a + c_{ab} & c_i^b = c_i^a + c_{ab} \\ path_i^b & = & path_i^a \cup \{a\} & \end{array} \quad (3)$$

Here all paths are updated with the arc cost c_{ab}. If the common cost u exceeds the upper bound (the allowed cost $C_{allowed}$) the item is dismissed. The degree of label b is the same or less than the degree of label a. For the next two cases, where label b is not empty, there are four possibilities during the insertion:

- The new item, inserted in label b, is not violating the property and it will be simply inserted at the right position.
- The new item is not better than any of the existing items in label b. It will be ignored.
- The position of the new item in label b, defined by the value of u, is the same as an existing item in label b, but the cost $c(i)$ of the new item is better than the cost of the stored item. In these cases, the existing item in label b will be overwritten.
- After the insertion of a new element in label b, one or more of the already existing items in label b violate a property of the PDPL. In this case, the violating items are deleted.

(iii) The degree of Label b is **not zero** and arc ab is **not part** of the SP

$$\begin{array}{rcll} \forall i | i \leq k_a : \quad u_{new}^b & = & u_i^a & c_{new}^b = c_i^a + c_{ab} \\ path_{new}^b & = & path_i^a \cup \{a\} & \end{array} \quad (4)$$

(iv) The degree of label b is **not zero** and arc ab is **part** of the SP $\forall i | i \leq k_a \wedge u_{new}^a + c_{ab} \leq C_{allowed}$:

$$\begin{array}{rcll} u_{new}^b & = & u_i^a + c_{ab} & c_{new}^b = c_i^a + c_{ab} \\ path_{new}^b & = & path_i^a \cup \{a\} & \end{array} \quad (5)$$

For both cases, a new item is inserted in label b, if it prevents the property of Equation (1). All items that are violating Equation (1) are deleted. For the description of the algorithm below, the update PDPL function will use the following notation:

$$PDPL(b).add(PDPL(a), u_{ab}, c_{ab}, a, C_{allowed})$$

$PDPL(b)$ is the label of node b that has to be updated. The parameters of the add-function are: $PDPL(a)$, a pointer to the label of the node where the arc starts, u_{ab}, the cost of the arc ab for a SP member, c_{ab}, the cost of the arc ab, the node a where the actual arc emanates and the allowed cost $C_{allowed}$. For a non SP arc u_{ab} is set to zero, for a SP arc it is set to c_{ab}.

III. Definition and Algorithmic Steps

The algorithm is based on the Suurballe algorithm with several added steps. A $PDPL$ is assigned to every node to store

the partially disjoint paths. Every time the algorithm permanently labels a SP node, the PDPL of the following SP nodes are updated. For the normal PDPL update and for the solution of a gap topology, special conditions are implemented.

Two conditions are necessary to force a restart of the algorithm. When a PDPL of a node, which is already permanently labelled, is changed, the algorithm is sent back to this node and executes the same paths again. This is necessary because the transformed network can contain cycles. If the transformed network looks like Figure 2 (OD-pair $a - c$, SP abc) the path $abdc$ can provide a reasonable partially disjoint path. In this case, the PDPL update of node c after performing node d will change, even if it is already permanently labelled. In such a situation, the algorithm is forced to step back to the node where this change occurred. It is done by unmarking all nodes until this node is reached.

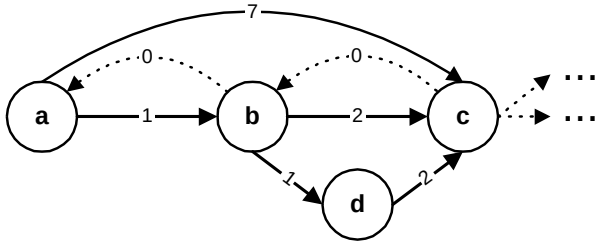

Fig. 2. Example Trap Topology – "Step Back"

In a gap topology the algorithm will stop when all nodes that are located before the gap are permanently labelled. It cannot find another minimum label. In this case, the cost of the following SP node is set to the worst node cost label $c(i)$ found so far. Now the algorithm can process the rest of the network. If the PDPL of this SP node is empty, the algorithm is terminating, because no solution exists.

The algorithm consists of three main parts, the initialisation step, the iteration step and the path generation step. For better understanding, the description of the algorithm is divided into several sections. First, the initialisation and the frame part of the algorithm are defined, then the three substeps are shown. The initialisation step is shown in Figure 3. The maintained variables have the following meaning: N is the set of network nodes, A is the set of network arcs, S is the set of SP nodes, L is the set of SP arcs and the sets B and R are part of a Boolean nodes label. Two different arc costs are used: the original arc cost c_{ij} and the reduced arc cost $c_{ij}^{'}$. The iteration step and the path collection step are depict in Figure 4. $A^{'}(i)$ is the set of emanating arcs from node i with an assigned reduced cost $c_{ij}^{'}$. The description begins with the first step.

Initialisation: The first three steps, the generation of the SP-tree routed at the origin node, the reduced cost transformation and the reversing of the direction of the SP arcs, are similar to the steps of the Suurballe algorithm. The SP-tree for the origin node is generated using the Dijkstra algorithm. The reduced cost transformation is applied with the node distance $c(i)$, which is a result of the SP algorithm. The SP arcs are excluded from this operation, because in a later step the untransformed SP arc cost is required. The "reverse the SP arc" function builds new arcs for the SP arcs with inverted directions and an assigned cost of zero. (One of the properties of the reduced cost transformation states that the cost of SP arcs after the transformation is zero.) In the following step, the $nextSP$ variable is set. The $nextSP$ variable points in a forward direction to the following SP node. The $nextSP$ variable of the origin node points to the second node on the SP, the $nextSP$ variable of this second node points to the third SP node and so on, until the destination node is reached. For non SP nodes this label points to a dummy node. (A dummy is a node that is not part of the network, but it is implemented as an additional node with a degree of zero.) The $nextSP$ variable provides two items of information. The first is the obvious next SP-node information, the second one indicates a SP node. The next substep within the initialisation is the calculation of the allowed cost. The next substep is the initialisation of the parameters for the Dijkstra part in the following iteration step. The Boolean node label is realised with the sets R and B. The cost label of the origin node is defined to be zero; the cost label of all other nodes is defined to be infinite. In the last substep the $PDPL(origin)$ is initialised. The last permanently labelled SP node $LastLabeledSPN$ is initialised with the $DummyNode$ and c_{max} is set to zero. c_{max} is the maximum distance label $c(i)$ set until now. This information is needed for the "restart at a gap" substep. On this transformed network, the iteration steps are applied.

Step 1 INITIALISATION

- Generate a SP-tree routed at the origin node.
- Apply the reduced cost transformation:
  $\forall(i,j)|(i,j) \in A \wedge (i,j) \notin L : c_{ij}^{'} = c_{ij} - c(j) + c(i)$
- Reverse SP-arc between the origin and the destination node:
  $\forall(i,j)|(i,j) \in L :$ build a new arc (j,i) with $c_{ji} = 0$
- Set the $nextSP$ variable:
  $\forall(i,j)|(i,j) \in L : nextSP(i) = j$
  $\forall i|i \notin S : nextSP(i) = 0$
- Calculate the allowed cost:
  $C_{allowed} = C_{SP}(1 - d)$
- Set the start parameter for the Dijkstra Part:
  $B = \{\} \quad R = N \quad \forall i|i \in R : c(i) \rightarrow \infty$
  $c(origin) = 0 \quad HEAP.insert(origin)$
- Set the start parameter for the PDPL:
  $PDPL(origin).insert(0, 0, origin)$
  $LastLabeledSPN = DummyNode$
  $c_{max} = 0$

Fig. 3. Algorithm - Initialisation Step

Iteration: The iteration step is similar to the iteration step of the Dijkstra algorithm with a heap data structure [9]. The differences are: The information of the predecessor variable is stored in the assigned $PDPL$ and the predecessor variable is not used, an additional label is maintained and the substeps are inserted. The label is processed in two ways. The information of the label is that a node is permanently labelled or not.

Step 2 ITERATION

while $HEAP \neq 0$ do
$HEAP.findMin(i)$
$HEAP.deleteMin()$
$STACK.insertTop(i)$
$B = B \cup \{i\}$ $R = R \setminus \{i\}$
SUBSTEP: UPDATE SP
$\forall(i,j)|(i,j) \in A^{'}(i)$:
SUBSTEP: UPDATE PDPL
If $(c(j) > c(i) + c_{ij}^{'})$ do
If $(c(j) > c_{max})$ do $c_{max} = c_j$
If $(c(j) = \infty)$ do
$c(j) = c(i) + c_{ij}^{'}$
$HEAP.insert(j)$
else
$c(j) = c(i) + c_{ij}^{'}$
$HEAP.decreasekey(j, c(j))$
SUBSEP: RESTART AT A GAP

Step 3 COLLECTION

- Chose from the $PDPL(destination)$ the path within the degree of divergence. If an inverted arc appears in the first set remove it from both sets.

Fig. 4. Algorithm - Iteration Step

The permanently marked nodes are stored in the $STACK$, but they are also marked with a conventional node label. This increases the memory usage of the algorithm, but it allows checking of the membership of a node in the $STACK$ in $O(1)$ time. The sequence of nodes stored in the $STACK$ is needed for the "update PDPL" substep. Unless the $HEAP$ is empty, the iteration is performed and the substeps are executed.

Path Generation: Every item in the $PDPL$, of the destination node, consists of a partially disjoint pre-solution path. The procedure used by the Suurballe algorithm is necessary to find the final solution "path pair". For every item a path pair is generated by taking the SP set and the pre-solution set of one $PDPL$ item. Every hop, which appears in the second set with a negative value, is discharged in both sets for this solution pair. The remaining arcs in both sets define the partially disjoint path pair. Which path pair of the solution set is the final solution depends on the interpretation of the degree of divergence. The cost of the two paths can be calculated by adding the original arc cost of the member arcs of the solution.

Now that the framework of the algorithm is known, the required substeps are defined and discussed. The substeps are depicted in Figure 5 for the "update the SP" substep, in Figure 6 for the "update the $PDPL$" substep and in Figure 7 for the "restart at a gap" substep.

Update SP: This step is executed when a SP node is permanently labelled. It starts at the actual node and updates the $PDPL$ assigned to the next SP node with the disjoint path information stored in the label of the first node. This is done from SP to SP node unless the $PDPL$ of the next SP node is not updated any more or the destination node is

SubStep UPDATE SP

if $(nextSP(i) \neq DummyNode)$ do
$a = i$
if (i is located on SP after LastLabeledSPN) do
$LastLabeledSPN = i$
while (no more PDPLs changed $\vee$
$nextSP(a) = DummyNode$) do
$b = nextSP(a)$
$PDPL(b).add(PDPL(a), c_{ab}, 0, a, C_{allowed})$
$a = nextSP(b)$

Fig. 5. Algorithm - Update SP Substep

reached. With every iteration, the commonly used cost of the items increases. Once the commonly used cost of all paths is higher than the allowed cost, no more paths are inserted in the $PDPL$. Note that the path cost variable c_{ij} in the "$PDPL$ add" function is set to zero. This is done because the reduced cost of the SP arcs equals zero, after the reduced cost transformation. The last permanently marked SP node is stored in the $LastLabeledSPN$ variable. The last permanently labelled SP node is the node, which is permanently labelled and the closest SP node to the destination node. In the programming solution this is realised with additional index information for the SP nodes. This information is needed by the "restart at a gap" substep.

SubStep] UPDATE PDPL

$PDPL(j).add(PDPL(i), 0, c_{ij}^{'}, i, C_{allowed})$
if $(j \in R \wedge PDPL(i)$ is changed) do
while $(j = STACK.getTop())$ do
$HEAP.insert(STACK.getTop())$
$STACK.deleteTop()$
$B = B \setminus \{i\}$ $R = R \cup \{i\}$
Start a new iteration of the main $while - loop$.

Fig. 6. Algorithm - Update PDPL Substep

Restart by Label Change: In this substep, for every arc processed, the PDPL of the terminating node is updated. If the PDPL of a node is changed, after this node is permanently labelled, a "step back" in the algorithm is necessary. The already processed nodes are removed from the $STACK$, inserted again in the $HEAP$ and unmarked. This is done unless the node, where the change of the PDPL occurred, is reached. This procedure forces the algorithm to start again at this position. When all needed nodes are unmarked, a new main while-loop iteration is started.

Restart at a Gap: The "restart at a Gap" substep is executed when the $HEAP$ is empty after the process of the main iteration step and the destination node is not permanently labelled. In such a case, the network has a gap topology. It is necessary to "step over" the gap. The next SP node, located after the

SubStep] RESTART AT A GAP
if $(HEAP = \{\} \wedge LastLabeledSPN \neq DestinationNode)$ do

$a = nextSPNode(LastLabeledSPN)$
if $(PDPL(a) \neq \{\})$ do
$c(a) = c_{max}$
$HEAP.insert(a)$

Fig. 7. Algorithm - Restart at a Gap Substep

last permanently labelled SP node on the SP, is a node after or within the gap. If the PDPL of this node is not empty, a partially disjoint path reaches the node. In this case the cost label $c(i)$ of this node, is set to the cost c_{max}, the maximum of all $c(i)$ labels except the labels with a infinite cost. To enable the start of the algorithm, this node is inserted in the $HEAP$. It works like a new pseudo origin node. The details of this situation are discussed in Section IV-A.

IV. DISCUSSION

This section discusses special conditions, correctness, the running time and possible further improvements of the algorithm.

A. Correctness

For the correctness discussion, two questions are addressed: Do the changes applied to the Suurballe algorithm influence the correctness of finding two completely disjoint paths? And secondly: Are all partially disjoint paths synthesised during the algorithm execution? In [8] the correctness of the Suurballe algorithm is proven. The changes made are:

Additional Label: The $PDPL$ results in no changes of the general procedure of the original algorithm. It remains passive during the execution of the algorithm and is described in detail in Section II. It has no influence on the correctness of the original Suurballe part of the algorithm.

Step Back: The step back procedure in the case of a PDPL change is necessary in a case of a trap topology. An example is depicted in Figure 2. The figure shows a graph after the application of the initialisation step. The SP is abc. The bold arcs are the original SP arcs with the assigned arc cost of c_{ij}. The cost of the remaining arcs is the reduced cost $c_{ij}^{'}$. The dashed arcs are SP arcs that were inserted in the reverse direction. During the execution of the algorithm, the original SP arcs (bold) are only processed by the "update SP" substep. Following the normal procedure the algorithm processes the permanently labelled node sequence of a, c, b, d. In this case, the partially disjoint path $abdc$ was not part of the $PDPL$ assigned to node c, when the node was permanently marked. The outgoing arcs of node c (dotted) were processed before this path was known. (If a SP node has only one incoming arc, it is processed after it is reached on a 'backwards' arc.) In this case, node c has to be processed again with the changed $PDPL$. The processing of node c has to make sure that all arcs and nodes are processed again. The order of the performed nodes is stored in the $STACK$. The algorithm can be forced to step back to node c by unmarking all nodes that were marked after node c. These nodes are inserted into the $HEAP$ again. The $PDPL$ and the node cost label $c(i)$ are not changed during this step. At the new start from node c, the before missing path, is now stored in the $PDPL(c)$ and all outgoing arcs can be further processed. Since the general network situation is not changed with this restart the permanently labelled node sequence is the same during the second run. Note that during the second run the "If $(c_j > c(i) + c_{ij}^{'})$ do" statement (Figure 4) in the iteration step is never true for nodes that are processed again. The programming solution uses an additional stack to process these nodes.

This step can be also seen as an application of a second run of the SP algorithm on the transformed network. Every time a permanently labelled node is changed, all permanently labelled SP nods are unmarked and inserted in the $HEAP$ again. Because the cost c_{ij} of the arcs remains the same and the general network structure is not changed, this has also no influence on the correctness of the SP algorithm. The algorithm will "walk" the same path again. A minimum of the node cost label $c(i)$ in the first run remains a minimum in the second run. The "restart" has no influence on the correctness of the underlying Suurballe algorithm.

Forward and Backward SP Arcs: The simultaneous processing of the forward SP arcs and the backward SP arcs for one SP node pair causes no problems. It is important to realise, that two different costs are maintained. The question is whether it could cause a cycle in one of the solution paths. A loop with this SP arcs results in a worse cost, because the backwards SP arc has a cost of zero and the forward arcs have a cost of $c_{ij} > 0$. See for example Figure 2 with the cycle: node $c-$dashed arc $cb-$node $b-$bold arc $bc-$node c. This path results in a cycle cost of $C_{cbc} \geq 0$. The node c is permanently labelled before the algorithm steps in the cycle and the cost, that the new path provides, is not better. So such a cycle causes no problem during the execution of the network algorithm.

Restart at a Gap: A gap divides the network into two subnetworks. The "restart at a gap" procedure initialises a second start of the original Suurballe algorithm after a gap. (One gap causes a division of the network into two subnetworks. Two gaps divide the network into three subnetworks and so on.) The two runs of the algorithm will find the partially disjoint path before and after the gap. The two solutions are "connected" with the $PDPL$, which provides the solution. The restart condition enables the execution of the algorithm after the gap, which causes no problems for the general correctness. The previously described method of doing this is that the SP nodes, located after the last permanently labelled SP node is initialised with a new node cost label. If this node provides no further paths (it is still part of the gap), in the next iteration, the next node on the SP is set. This is done unless the algo-

rithm "steps over" the gap or the SP nodes have no PDPL entries any more. In this case, the algorithm terminates with no solution. The new cost is set to the highest cost label used before the gap, to avoid interference between the subnetworks. If the cost is set to the highest occurring cost label $c(i)$, all arcs that reach a node that is part of the subnetwork, before the gap, will result in a higher or equal cost to the already set cost label. It equals an "offset" for all node cost labels $c(i)$. The separate cost label $c(i)$ is needed in this case, in all other cases the entry of $c(i)$ is similar to the cost c_k^i of the bottom item in the $PDPL(i)$. The algorithm will find two disjoint path pairs, one before the gap topology the second one after the gap topology. This causes also no problem.

The discussion showed that the changes applied to the algorithm have no influence on the complete disjoint path generation. The remaining question is if the algorithm finds and stores all partially disjoint paths in the $PDPL$. One result of the observation in [6] was that shareable paths that result in a better cost, are SP arcs. This is implemented by the "update SP" substep. The shareable arcs are performed at the head of a main while-loop. During the execution all SP arcs are processed. Once an arc reaches the PDPL of a SP node the PDPL of the SP nodes are updated. Once the algorithm reaches such a node, all partially disjoint paths are stored in the PDPL. One shared SP path divides the network into two subnetworks: the subnetwork before the shared path and the subnetwork after the shared path. All partially disjoint path situations can be reduced to the combination of such subnetworks. For example two not connected shared paths divides the network a subnetwork before the first shared path and a subnetwork after the shared path. The second SP network itself is divided by the second shared path in a before and after subnetwork. The first question, which has to be addressed, is if the SP for the transformed network is found in the first subnetwork. By definition, the Dijkstra algorithm will find the SP from the origin node to all network nodes; also to the first node of the shared path part. The same is true for the second subnetwork, because the algorithm marks the tail node of the shared path as permanent at one stage and is searching then from this node the shortest route to the destination node. This covers the SP generation for the second subnetwork. This is certainly only true if the shortest path in the transformed network is using this nodes, but otherwise an other path is better. When the shared path tail node is permanently labelled, all partially disjoint paths are known by the PDPL they are also processed to the destination node. Otherwise at one stage of the execution a restart is initialised and after that all paths are known. Therefore all best partially disjoint paths are found.

B. Algorithmic Performance

The worst case analysis of the algorithmic performance produces the upper bound of $O(n^3)$. This worst case running time is mainly due to the unlikely assumption that the shortest path consists of all network nodes. For a practical network, the assumption of a constant SP-node-length is more probable. The running time in this case is determined by the performance of the Dijkstra implementation. For the used implementation this yields $O(m \cdot \log_{\frac{m}{n}} n)$.

V. Conclusion

This paper has described an algorithm for solving the shortest partially disjoint path pair problem. The solution is extended to solve the all-best partially disjoint path pairs between an OD-pair problem without major changes of complexity in the calculations. For a degree of divergence of $d = 0\%$ all possible best paths are found, between the complete disjoint path pair and two times the shortest path. The presented algorithm has a expected worst case running time of $O(n^3)$. With the assumption, that the average SP node number is independent of the network parameters, the complexity is reduced to the time consumption of an efficient implementation of the Dijkstra algorithm.

An extended version of this paper [11] includes an example to illustrate the procedure, a detailed discussion of the algorithmic performance and possible further improvements.

VI. Acknowledgements

The helpful comments and discussions by the colleagues at the Centre for Advanced Technology in Telecommunications (CATT), in particular Sayaan Choudhury, are gratefully acknowledged. The authors would also like to thank Andreas Christ (University of Applied Science Offenburg Germany) for his support.

References

[1] D. Torrieri, "Algorithms for finding an optimal set of short disjoint paths in a communication network," *IEEE Transactions on Communications*, vol. 40, no. 11, pp. 1698–1702, 1992.

[2] R. G. Ogier, V. Rutenburg, and N. Shacham, "Distributed algorithms for computing shortest pairs of disjoint paths," *IEEE Transaction on Information Theory*, vol. 39, no. 2, pp. 443–455, 1993.

[3] A. Iwata, R. Izmailov, and B. Sengupta, "Alternative routing methods for pnni networks with partially disjoint paths," *IEEE Globecom -New York- 1998*, vol. 1, pp. 621–626, 1998.

[4] A.W. Brander and M. C. Sinclair, "A comparative study of k-shortest path algorithms," *Proc 11th UK Performance Engineering Workshop, Liverpool*, pp. 370–379, 1996.

[5] S. Choudhury, *Approaches for Robust Virtual Path Synthesis in ATM Networks*, Ph.D. thesis, Department of Computer Systems Engineering, RMIT University, Melbourne, July 1999, Thesis submitted for the degree of Doctor of Philosophy.

[6] Alexander A. Kist and R.J. Harris, *Note on the Problem of Partially Disjoint Paths in a Communication Networks*, Royal Melbourne Institute of Technology Melbourne, Australia, October 2002, To appear.

[7] J.W. Suurballe, "Disjoint paths in a network," *Networks*, vol. 4, pp. 125–145, 1974.

[8] J.W. Suurballe and R. E Tarjan, "A quick method for finding shortest pairs of disjoint paths," *Networks*, vol. 14, pp. 325–336, 1984.

[9] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[10] E.W. Dijkstra, "Note on two problem in connection with graphs," *Numerische Mathematik*, pp. 269–271, 1959.

[11] Alexander A. Kist and R.J. Harris, *A Heuristic to Generate All Best Partially Disjoint Paths in a Communication Networks*, Royal Melbourne Institute of Technology Melbourne, Australia, January 2000, Technical Report.