平成25年度　修士論文

リンク故障を考慮した
負荷分散IPルーチング方式

所属 ：　情報・通信工学専攻
情報通信システムコース
大木研究室
提出者 ：　1231080 本間 奨
主任指導教員：　大木英司 教授
指導教員：　來住直人 教授
提出日 ：　2014年 1月 27日

専攻主任印	主指導教員印	指導教員印

概要

インターネット需要が増えたことで，ネットワーク内のトラヒックが急速に増大し，輻輳や通信遅延を引き起こしている．Internet Protocol (IP) ネットワークにおけるそれらの対策としてはトラヒックが一部のリンクに集中しないような経路制御を行うことである．ネットワークにおいて適切な経路制御を行うことはネットワーク資源の有効利用やスループットの改善に繋がる．またネットワークにおいて故障が発生した際に，多くのサービスやユーザに大きく影響を与える．この影響を小さくすることでもネットワーク資源は有効に活用できる．本論文では，リンク故障が発生してもトラヒックを分散し，ネットワークの負荷分散を行う方式を提案する．過去に提案された負荷分散方式は，リンク故障が発生していない場合に，トラヒックを分散して負荷分散を行い，高い性能を示しているが，リンク故障が発生しネットワークが復旧するまでにおいては性能が保証されていない．提案方式ではリンク故障が発生した時においてもトラヒックの分散処理を行い，負荷分散を行う．考えられる方式として，リンク故障が発生した時に新たにすべてのリンクの分散比を再計算して割り当てを行う方式である全体最適化方式がある．しかし，すべてのリンクに対して再計算を行うと，復旧を考えた場合に効率が悪い．そこで，故障したリンクを経由する発着ノードにおいてのみ分散比を新たに再計算する局所分散方式を提案する．この方式により，分散比を再計算する箇所を少なくすることで，再計算および再割り当ての効率が向上すると思われる．シミュレーションを用いてルーチング性能を評価した結果，ネットワーク輻輳率の低減においては，局所分散方式は全体最適化方式とほぼ同等の性能を示し，計算時間においては局所分散方式が全体最適化方式より良い性能を有していることを示す．

目 次

第1章　序論

1.1　研究背景

近年, インターネットの発達により，トラヒックは年々増加し続けている．それに伴いネットワーク資源の使用率やスループットの向上，輻輳の回避を目的として，ルーチングを適切に制御する必要がある．ルーチング制御性能の指標として，ネットワーク輻輳率がある．ネットワーク輻輳率とはトラヒックの流れるネットワークの中で最も混雑しているリンクの使用率のことである．このネットワーク輻輳率を低減させることによって，輻輳を回避し，より多くのトラヒックをネットワークに流すことができる．

ネットワーク輻輳率を低減させるようなルーチング方式は [1]-[2] によって研究されている．Wang ら [1] は，一般的なトラヒック制御問題の定式化を行った．すなわち，発/着ノードの間で自由にトラヒック要求を分散できる問題として定式化を行った. この複雑な経路制御は Multi Protocol Label Switching (MPLS) や，Traffic-Engineering (TE) 技術を用いて実現することができる．しかし，現在のネットワークは Open Shortest Path First (OSPF) や, Intermediate System to Intermediate System (IS-IS) のような最短経路のプロトコルが主流である．よって, 既に導入されている IP ルータに MPLS-TE を採用する為には，装置のアップグレードを行う必要がある．現存するすべての IP ルータに装置のアップグレードを行うのは現実的ではない．そのため OSPF には，ネットワークに最適なリンクコストを設定して経路制御を行う方法が研究されている．OSPF ではすべてのパケットはリンクコストを基準とした最短経路で転送されるため，最適なリンクコストを設定することにより，経路制御を行うことができる．しかし，トラヒック需要に変化が生じた場合はこのリンクコストを再計算し，ネットワーク管理者はそれをネットワークに再設定する必要がある．再計算されたリンクコストの再設定を行う頻度が高くなると，再設定を行っている IP ルータ同士で経路情報の不一致が起こり，予期せぬルーチングが行われネットワークが不安定になり，パケットの消失やループの発生を引き起こしてしまう原因となる．

そこで，リンクコストを変更することなく現在の OSPF ネットワークに対応できる Smart-OSPF(S-OSPF) が Mishra らによって提案された [3]．この S-OSPF では発エッジノードにおいて，トラヒックを隣接ノードへ分散して転送する．また，それぞれ隣接ノードへ転送されたトラヒックは，OSPF プロトコルによって決定された最短経路にしたがってトラヒックが転送される．S-OSPF では，トラヒックは発ノードからその隣接ノードにのみ分散される．すなわち，自らが発ノードとなる場合にのみ，トラヒックを分散させれば良いので，実装において MPLS-TE のように大掛かりな変更を行う必要がない．したがって，S-OSPF の実装は MPLS-TE より容易である．
[3]，[4] で提案されている S-OSPF では，各エッジノードがトラヒックを中継するための通常のルーチングテーブルと，経路制御のための最適な分散比を持った特殊なルーチングテーブルを持つ必要がある．しかし，この S-OSPF はリンク故障に対応しておらず，リンク故障が発生した場合においては通常の OSPF で動作し，従来と同様にネットワーク輻輳率が上昇してしまうと考えられる． そこで本研究ではリンク故障が発生しても通常時と同じような分散を行い，ネットワークの輻輳を回避できる経路制御方式を提案し，その評価を行う．

1.2　研究の目的と論文構成

本研究では，ネットワークの輻輳の回避を目的とし，リンク故障が発生してもトラヒックを分散し，ネットワークの負荷分散を行う方式を提案する．本論文の構成は, 第 2 章で関連研究とその問題について述べ, 第 3 章では, ネットワークモデルの説明を行い，第 4 章で提案方式の説明と定式化について述べる. 第 5 章では性能評価を行い，第 6 章でまとめを述べる.

第2章　関連研究

2.1　最短経路ルーチング

IP ネットワークにおいてパケットを送るとき経路を決める手段としてルーチングプロトコルが用いられる．現在の主要なルーチングプロトコルとして Open Shortest Path Fast(OSPF) がよく使われていることが知られている．IP ネットワークの各リンクにはリンクコストが与えらえている．リンクコストは通常，帯域幅の逆数に比例した値が自動的に与えられる．したがって帯域の大きなリンクほど小さなリンクコストとなる．このリンクコストはネットワーク管理者によって任意の値に変更することも可能である．OSPF でのルーチングはこのリンクコストの和が最小となるような最短経路を選択して送られる．目的地ノードへ最短経路でトラヒックを転送するために各ノードは目的地ノードに対する次ホップが記述されるルーチングテーブルを有し, 次ホップをテーブルから参照することで転送を行っている．OSPF は各ノードが目的地ノードに対して定められた次ノードにパケットを送ることで１つの決まった最短経路でパケットが転送出来るのである．

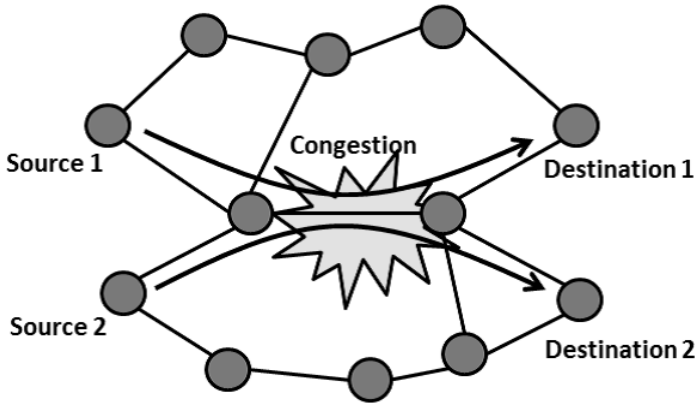

図 2.1: 最短経路ルーチング:OSPF

この OSPF のようなリンクステート型ルーチング方式では目的地ノードまでのリンクコストの総和が最小となる経路が選択される．最短経路が一意に決まってしまうために最短経路上のリンクにトラヒックが集中してしまうということが問題視されている．また OSPF の特性として，最短経路上のリンクが故障してしまったとき最短経路の再計算が各ルータで自動的に行われる．この際に複数のルータの間のルーチング情報の不一致により，予期せぬルーチングがなさせるためネットワークは不安定になることがある．本論文で不安定な状態とは目的地までパケットが到着しない状態を言う．ループ現象がその一例である．ループ現象とはパケットが１度通ったルータに再び戻って来てしまう現象である．これはあるルータの計算が完了しているにもかかわらず，他のルータの計算が終わっていないときにこの現象が発生することがある．

先行研究でトラヒック需要が変わるたびに適切なリンクコストを設定することで輻輳を回避する経路制御の研究が紹介された．しかしながら，この方式ではトラヒック需要が変化に対して，その都度最適なリンクコストの計算しなくてはならず，さらにリンクコストの変更が行われ, ルータの最短経路の再計算が頻繁に行われるためにかえってネットワークが不安定になる恐れがある．

2.2 Smart-OSPF

ネットワークが不安定になるという問題から，リンクコストの変更をせずに与えられたトラヒック需要からネットワークの輻輳を防ぐことができる経路を予め計算し，最適なルーチングが可能となる方式が検討されている．また経路が一意に定まらないようにノードでトラヒックをある割合で分散させる手法が研究されている．その中の技術の１つに Smart OSPF（S-OSPF) という方式がある [3]．この方式では各発エッジノードにおいてのみトラヒックを分散し，中継ノードでは OSPF プロトコルによる最短経路に従ってトラヒックを転送させる．この時，発ノードから中継ノードへトラヒックを分散する比率は線形計画問題を解くことによって決定される．図 2.2 は S-OSPF の概略を示したものである．これによってネットワークの特定のリンクでの輻輳を避け，ネットワーク資源を有向に活用することが可能となる．

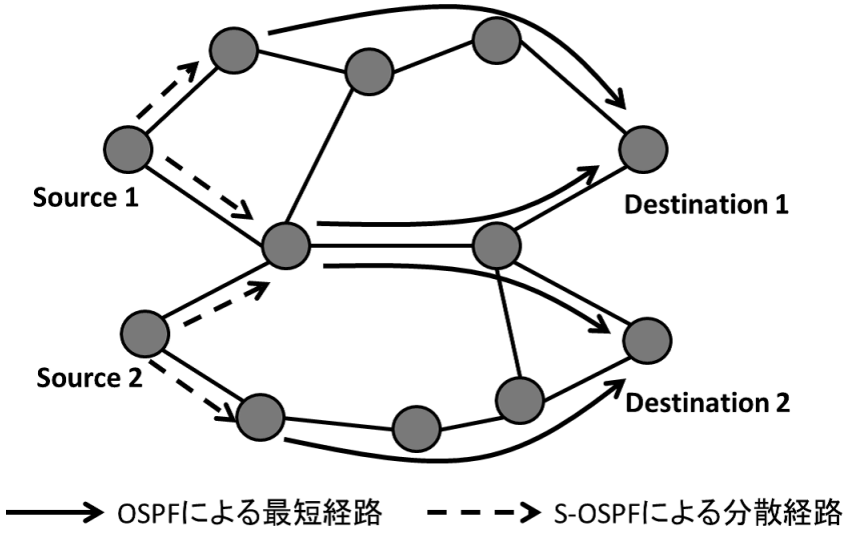

図 2.2: S-OSPF

S-OSPF では，各エッジノードがトラヒックを中継するための通常のルーチングテーブルと，経路制御のための最適な分散比を持った特殊なルーチングテーブルを持つ必要がある．split S-OSPF は最適な分散比を用いて分散を行うので輻輳率の低減は最も良いとされる．

しかし，この S-OSPF はリンク故障に対応しておらず，リンク故障が発生した場合においては通常の OSPF で動作すると予想され，従来と同様にネットワーク輻輳率が上昇してしまうと考えられる．

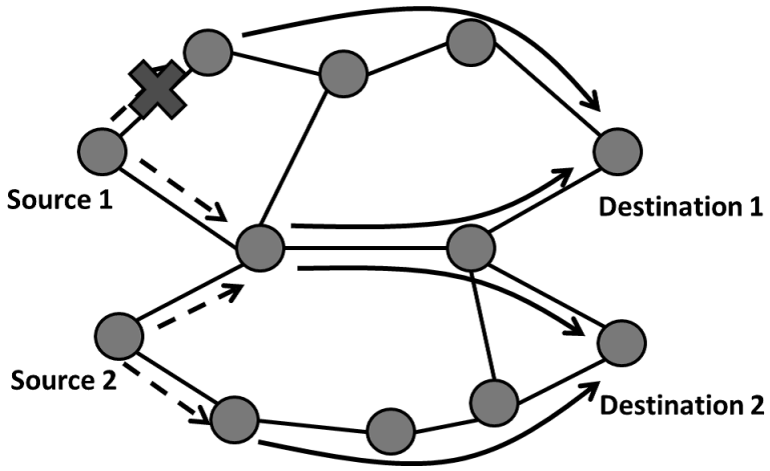

図 2.3: リンク故障における S-OSPF

そこで，リンク故障が発生しても S-OSPF が問題なく機能するように分散が行え，通常時のようにネットワーク輻輳率を低減させることができるような方式を考える必要がある．

第3章　ネットワークモデル

ネットワークモデルは，有向グラフ$G(V,E)$で表される．Vはノードの集合，Eはリンクの集合である．また，$Q \subseteq V$である．Nは，ノード数である．ノード$i \in V$からノード$j \in V$までのリンクを$link(i,j) \in E$とする．x_{ij}^{pq}はノード$p \in Q$からノード$q \in Q$のトラヒックで$link(i,j) \in E$を通るトラヒックの割合である．c_{ij}は，$link(i,j) \in E$の容量である．$link(i,j)$を通過する総トラヒック量はy_{ij}で表される．$T = \{d_{pq}\}$は，トラヒック需要を表す3次のトラヒック行列であり，d_{pq}は、ノードpから，ノードqまでのトラヒック需要である．mは転送を行う隣接ノードの数とし，ネットワーク輻輳率をrとする．ネットワーク輻輳率はネットワークにおいてもっとも利用率が高いリンクの値であり、以下のように定義される．

$$r = \max_{(i,j) \in E} \frac{y_{ij}}{c_{ij}} \tag{3.1}$$

ネットワークに新たに投入可能なトラヒック総量は$\frac{1-r}{r} d_{pq}$である．そのネットワークにトラヒック需要d_{pq}を流しきったとき，トラヒック総量は$\frac{1}{r} d_{pq}$となり，そのときのネットワーク輻輳率は1になる．ネットワーク輻輳率が1になるということはそれ以上トラヒックを投入できない．したがって多くのトラヒックを新たに投入したければトラヒック総量は$\frac{1-r}{r} d_{pq}$を最大にすればよい．これはrを最小化することと同義である．このネットワーク輻輳率の低いネットワークは性能の高いネットワークと言える．本研究においてネットワーク輻輳率rをルーチング指標とする．これを減少させることを目的とする．

ここで，[3]で示したS-OSPFでは，発ノードp，着ノードqにおけるトラヒック要求d_{pq}の集合であるトラヒック行列$T = d_{pq}$が既知であると仮定している．このようにトラヒック行列が既知であるトラヒックモデルのことをパイプモデルと言う．しかし，ネットワーク管理者にとって実際のトラヒック行列を測定，予測することは，非常に困難であることが知られている[5, 6]．トラヒック行列を測定するためには，発ノードにおいて着ノードアドレスを確認して，着ノード毎にパケット長とパケット数を確認する必要がある．この処理は，伝送速度が高速であるほど非常に大きな計算機パワーを必要とする．それに加えて，発/着ノード間のトラヒック需要は頻繁に変動するため，正確なトラヒック行列を測定することは困難である．

実際のトラヒック行列を測定するのは非常に困難であるが，ある時間における，ノードへの流入，ノードからの流出トラヒックを測定することは容易である．実際のトラヒック行列は不明であるが，各ノードにおける流入/流出トラヒックが既知であるトラヒックモデルとして，ホースモデルが知られている[7]．ホースモデルでは，正確なトラヒック行列$T = \{d_{pq}\}$が与えられない代わりに，d_{pq}の境界が式3.2a-式3.2bで与えられる．ここで，α_pはノードpがネットワークに送り込むことができる最大のトラヒック量，β_qはノードqがネットワークから受け取ることのできる最大トラヒック量，Qはエッジノードの集合である．

$$\sum_{q \in Q} d_{pq} \leq \alpha_p \quad p \in Q \tag{3.2a}$$

$$\sum_{p \in Q} d_{pq} \leq \beta_q \quad q \in Q, \tag{3.2b}$$

ホースモデルの場合，トラヒック行列が既知である場合と異なり，ネットワーク輻輳率を一意に決定することができない．そのため，ホースモデルにおいては，起こりうるすべてのトラヒックパターンを考え，その中で最も高いネットワーク輻輳率を，ネットワーク輻輳率として扱う．

第4章　提案方式

4.1　概要

定常時に輻輳回避を可能にし，故障時においても輻輳が上がりにくい負荷分散方式を提案する．S-OSPF においてリンク故障に対応する最も簡単な方法としては, 故障したら全体を再度分散しなおす方法がある．しかし，一つのリンク故障に対して全体の分散を再度計算し割り当てるという対応は，復旧を考えた場合に効率が悪いと考えられる. そこでリンク故障を起こした時に，再度計算するリンクの数を少なくするため, リンク故障に隣接しているノードにおいてのみ分散を考える局所分散方式を考案した.

4.2 全体最適化方式

全体最適化方式は，リンク故障が発生した時に，全体の分散比を新たに再計算して割り当てる方式である．すなわち，リンク故障が起きたリンクを除いたすべての発着ノードにおいて分散比を再計算する．この方式は故障リンクを除いた状態での最低となる輻輳率を求めるので，故障時における全体の輻輳率の低減は最も良いと考えられる．

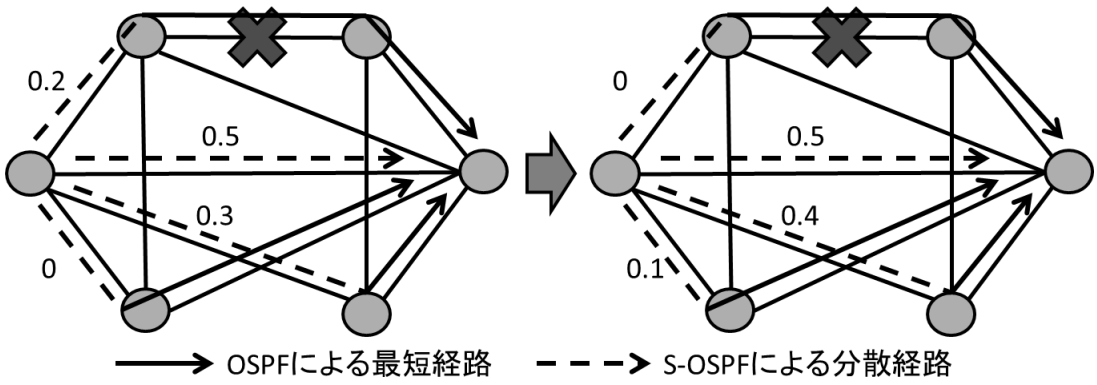

図 4.1: 全体最適化方式

この方式は，単純にリンク故障が発生したリンクを取り除いて再度 S-OSPF を求めると同義なので，式も S-OSPF と同じ式で表すことができる．S-OSPF を表す式を以下式 4.1a-式 4.1f で示す．

$$\min r \tag{4.1a}$$

$$s.t. \quad x_{ij}^{pq} - \sum_{j\prime:(j\prime,i)\in E} x_{j\prime i}^{pq} = 0$$

$$p,q \in Q, i \neq p, i \neq q, j = OSPF_{nexthop_i^{pq}} \tag{4.1b}$$

$$\sum_{j:(i,j)\in E, j\neq OSPF_{ancestor_i^{pq}}} x_{ij}^{pq} - \sum_{j\prime:(j\prime,i)\in E} x_{j\prime i}^{pq} = 1$$

$$p,q \in Q, i = p \tag{4.1c}$$

$$\sum_{p,q\in Q} d_{pq} x_{ij}^{pq} \leq c_{ij} \cdot r \quad (i,j) \in E \tag{4.1d}$$

$$0 \leq x_{ij}^{pq} \leq 1 \quad p,q \in Q, (i,j) \in E \tag{4.1e}$$

$$0 \leq r \leq 1 \tag{4.1f}$$

しかし全体の分散比を再計算するということは，故障したリンクを経由しない，まったく関係のない発着ノード対においても再計算を行う．全体最適化を計算するのに必要な計算量はノード数 N において $O(N^8)$ オーダーである．また，リンク故障の多くは早く復旧する可能性がある [8]．リンク故障の 80 %は 10 分以内に復旧し，そしてその中の 50 %は 1 分以内に復旧する．そのため，復旧を考慮すると全体の分散比を書き換えるのは効率が悪いと考えられる．従って我々は，リンク故障に関係する発着ノードにおいてのみ分散を考える局所分散方式を提案する．

4.3 局所分散方式

局所分散方式は，リンク故障が発生したときに，リンク故障に関係した発着ノードのみに対して分散比を再計算する方式である．故障リンクを経由する発着ノードに対しては全体最適化と同様に分散比の再計算を行うが，故障リンクを経由しない発着ノードに対しては分散比を変更しない．従って，全体最適化方式と比較すると輻輳率が上昇する可能性がある．しかし復旧を考慮すると，再計算および分散比の再設定を行う箇所を削減しているため，全体最適化方式より，計算速度や再設定箇所の面で優れている．

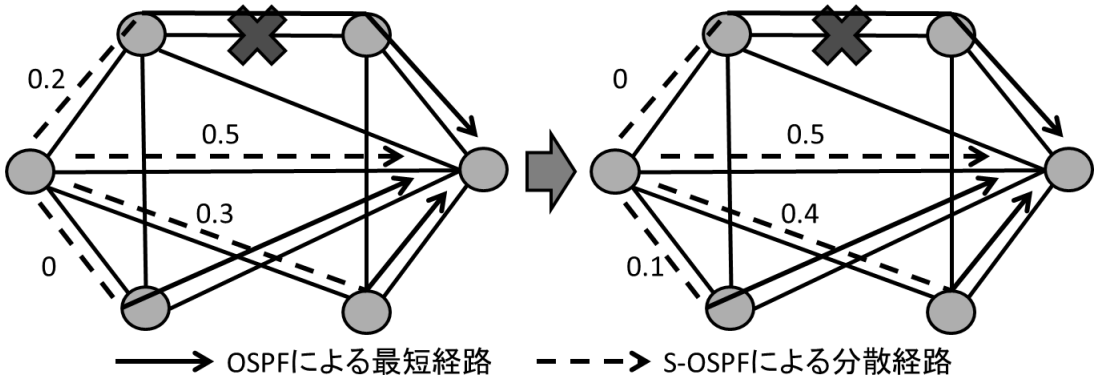

図 4.2: 故障リンクを経由する場合

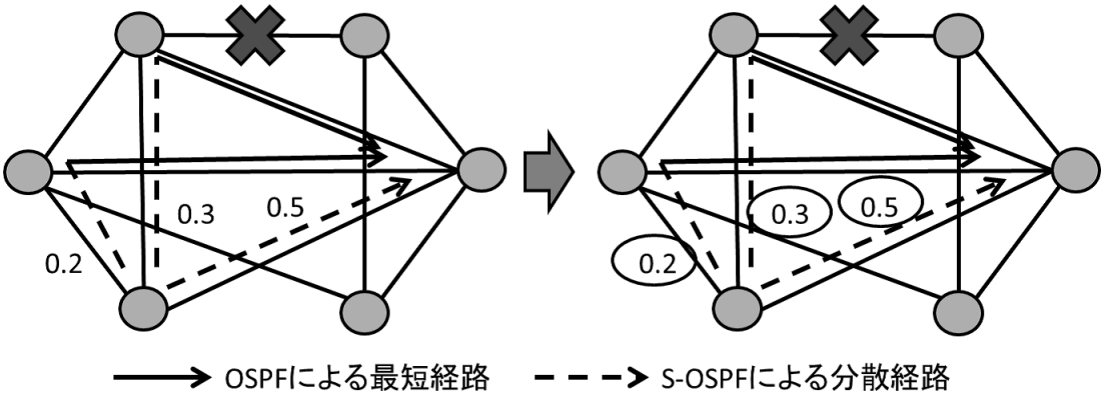

図 4.3: 故障リンクを経由しない場合

この方式は，S-OSPF から故障したリンクを経由しない発着ノードの制約式を固定することで表現できる．この S-OSPF を表す式を以下 4.2a-式 4.2g で示す．

$$\min r \tag{4.2a}$$

$$s.t. \quad x_{ij}^{pq} - \sum_{j\prime:(j\prime,i)\in E} x_{j\prime i}^{pq} = 0$$

$$p,q \in Q, i \neq p, i \neq q, j = OSPF_{nexthop_i^{pq}} \tag{4.2b}$$

$$\sum_{j:(i,j)\in E, j\neq OSPF_{ancestor_i^{pq}}} x_{ij}^{pq} - \sum_{j\prime:(j\prime,i)\in E} x_{j\prime i}^{pq} = 1$$

$$p,q \in Q, i = p \tag{4.2c}$$

$$\sum_{p,q\in Q} d_{pq} x_{ij}^{pq} \leq c_{ij} \cdot r \quad (i,j) \in E \tag{4.2d}$$

$$x_{ij}^{pq} = X_{ij}^{pq} \quad (i,j) \in E i \neq i_b, j \neq j_b, \tag{4.2e}$$

$$0 \leq x_{ij}^{pq} \leq 1 \quad p,q \in Q, (i,j) \in E \tag{4.2f}$$

$$0 \leq r \leq 1 \tag{4.2g}$$

ここで式 4.2e にある X_{ij}^{pq} とは，リンク故障が発生していない通常時の分散比であり，また i_b と j_b はリンク故障が発生している $link(i,j)$ である．

第5章　性能評価

5.1　サンプルネットワークを用いた性能評価

故障時における局所分散方式の性能を，故障時における OSPF，全体最適化方式の性能と比較する．LP ソルバとして CPLEX[9] を用い，性能指標はネットワーク輻輳率 r とする．対象とするネットワークトポロジは [3] で用いられたネットワークである図 5.1 をサンプルネットワークとして用いた．また，ネットワークのすべてのノードはエッジノードであるとする．リンク容量及びトラヒック需要はランダムに生成して与えた．各方式のリンクコストはリンク容量に基づき，大容量なリンクほどリンクコストが低くなるように設定した．また，故障時とは，各リンクを 1 つ壊して通れなくする単一故障とする．

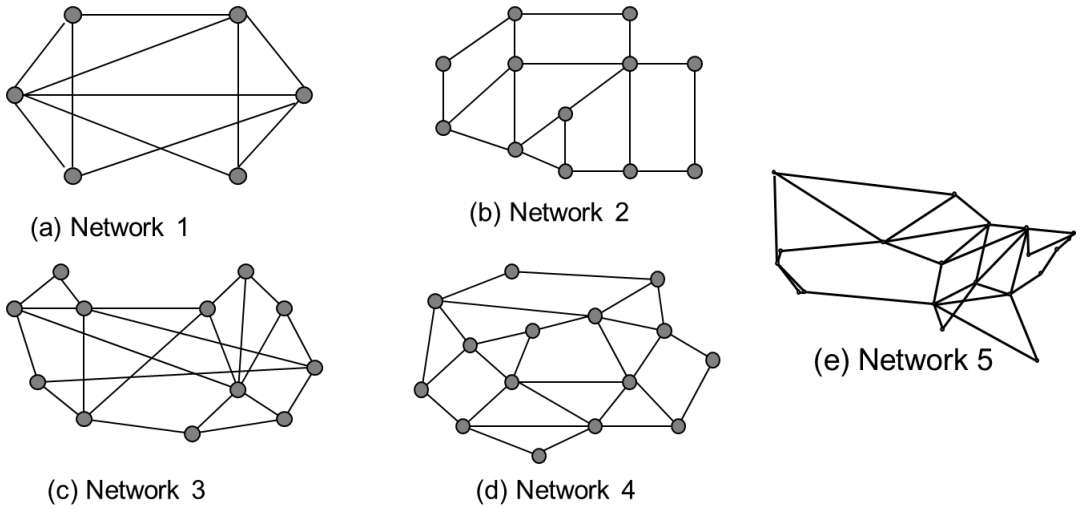

図 5.1: サンプルネットワーク

表 5.1: ネットワークの特性

Network type	No. of nodes	No. of links	Average node degree
Network 1	6	24	4.00
Network 2	12	36	3.00
Network 3	12	48	4.00
Network 4	15	56	3.73
Network 5	20	68	3.40

表 5.2，図 5.2 にパイプモデルにおける通常時の輻輳率を，表 5.3，図 5.3 に故障時のネットワーク輻輳率を示す．異なる方式間でネットワーク輻輳率を比較するため，通常時では OSPF で，故障時は再経路計算した OSPF で正規化した．また，故障時の輻輳率はリンク 1 つを故障させた時の輻輳率を，すべてのリンクで試行し，その平均値を用いた．

表 5.2: 通常時のネットワーク輻輳率

Network type	r OSPF	r S-OSPF
(a)	1.000	0.636
(b)	1.000	0.791
(c)	1.000	0.707
(d)	1.000	0.534
(e)	1.000	0.780

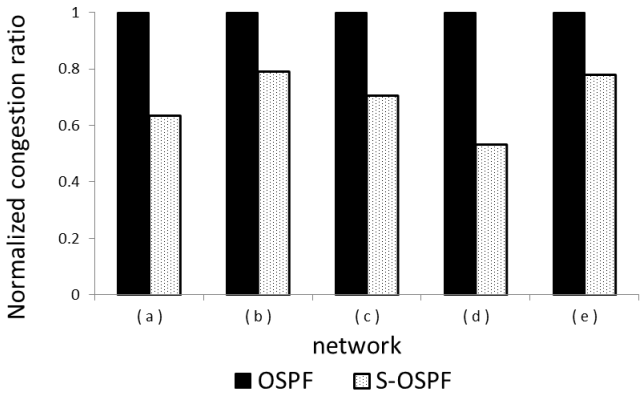

図 5.2: 通常時の輻輳率の比較

表 5.3: 故障時のネットワーク輻輳率

Network type	r OSPF	r 全体最適化	r 局所分散
(a)	1.000	0.691	0.737
(b)	1.000	0.819	0.882
(c)	1.000	0.686	0.784
(d)	1.000	0.608	0.669
(e)	1.000	0.809	0.833

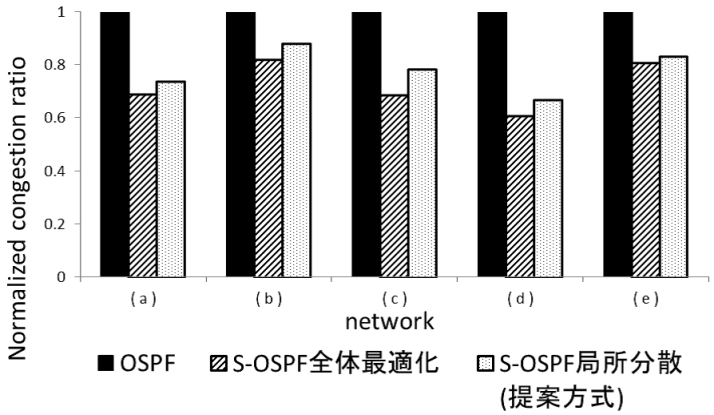

図 5.3: リンク故障時の輻輳率の比較

結果より，局所分散は故障時の OSPF に対して約 15 %から 30 %程度ネットワーク輻輳率が低減できた．全体最適化に対しては，ネットワーク輻輳率は高くなるものの，その差が 5 %から 10 %程度である．

次に，ホースモデルにおいての通常時のネットワーク輻輳率を，表5.4，図5.4に，故障時のネットワーク輻輳率を表5.5，図5.5に示す．ホースモデルにおいてもパイプモデルと同様，通常時ではOSPFで，故障時は経路再計算したOSPFで正規化した．

表 5.4: ホースモデルの通常時のネットワーク輻輳率

Network type	r OSPF	r S-OSPF
(a)	1.000	0.356
(b)	1.000	0.717
(c)	1.000	0.480
(d)	1.000	0.456
(e)	1.000	0.656

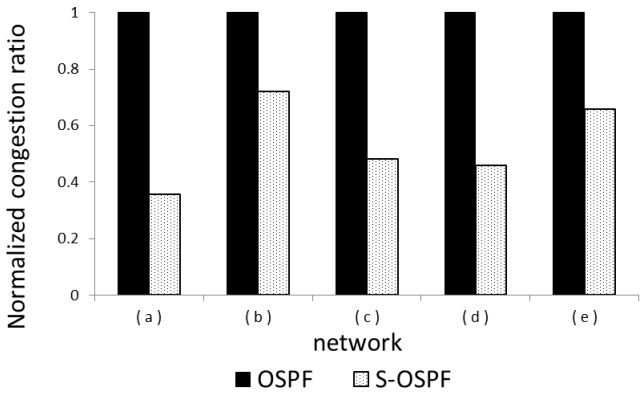

図 5.4: ホースモデルの輻輳率の比較 (通常時)

表 5.5: ホースモデルの故障時のネットワーク輻輳率

Network type	r OSPF	r 全体最適化	r 局所分散
(a)	1.000	0.461	0.474
(b)	1.000	0.474	0.495
(c)	1.000	0.341	0.362
(d)	1.000	0.281	0.298
(e)	1.000	0.412	0.413

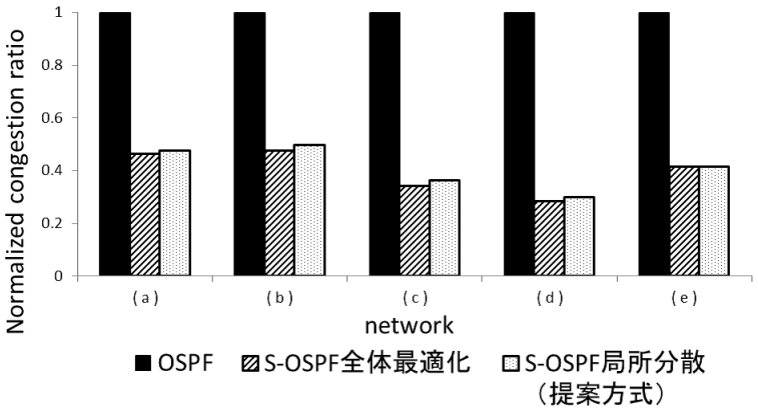

図 5.5: ホースモデルの輻輳率の比較 (故障時)

結果より，経路再計算したOSPFに対しては約50 %から65 %ほどネットワーク輻輳率を低減できた．これはホースモデルの影響により，OSPFのネットワーク輻輳率

が上がったためだと考えられる．全体最適化に対してはほとんど差が見られない．すなわちホースモデルにおいては局所分散は全体最適化とほぼ同等の性能を持つことが分かる．

5.2 ランダムトポロジを用いた性能評価

提案方式のトポロジに対するネットワーク輻輳率の変化を定量的に評価するため，ランダムにネットワークを作成し評価した．評価ネットワークはネットワークトポロジジェネレータ BRITE[10] を用いて作成した．ネットワークの規模として，ノード数 N10，20，30，40 でそれぞれネットワークを作成した．作成したネットワークトポロジの特性を図 5.6 に示す．

表 5.6: ランダムトポロジの構成

No. of nodes	No. of links	Average node degree
N : 10	40	4.00
N : 20	80	4.00
N : 30	120	4.00
N : 40	160	4.00

このランダムトポロジを用いて各方式のネットワーク輻輳率を比較した．表 5.7，図 5.6 にパイプモデルにおけるランダムトポロジの輻輳率を示す．

表 5.7: 各方式のネットワーク輻輳率

Network type	r OSPF	r 全体最適化	r 局所分散
N 10	1.000	0.771	0.832
N 20	1.000	0.766	0.816
N 30	1.000	0.667	0.736
N 40	1.000	0.621	0.681

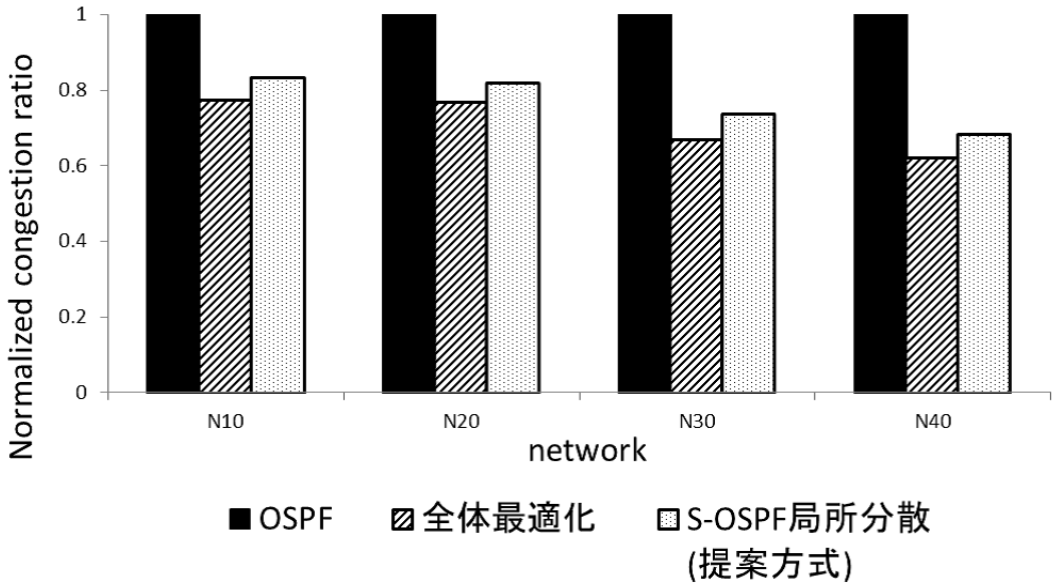

図 5.6: ランダムトポロジのネットワーク輻輳率

結果より，ランダムトポロジにおいても，同様に故障時の OSPF に対してネットワーク輻輳率を低減することができた．また，ネットワーク規模が大きくなるにつれ，ネッ

トワーク輻輳率が下がってきている．これはノード数が多くなることで同じリンクを経由しなくなるからであると考えられる．

次に，ホースモデルにおけるネットワーク輻輳率を表 5.8，図 5.7 に示す．

表 5.8: ホースモデルにおける各方式のネットワーク輻輳率輻輳率

Network type	r OSPF	r 全体最適化	r 局所分散
N 10	1.000	0.425	0.517
N 20	1.000	0.441	0.505
N 30	1.000	0.579	0.696
N 40	1.000	0.466	0.533

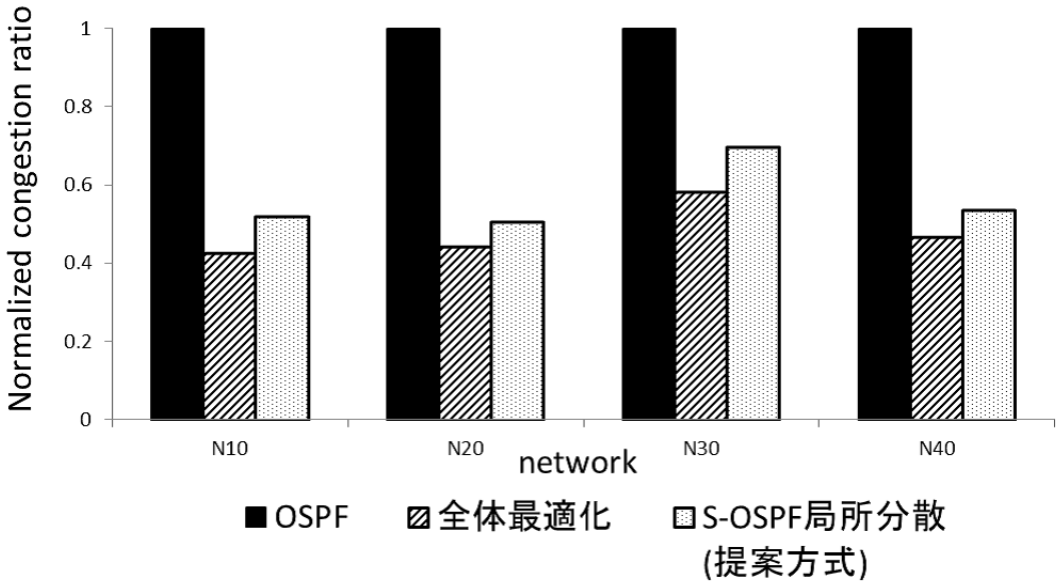

図 5.7: ホースモデルにおけるランダムトポロジのネットワーク輻輳率

結果より，パイプモデルと同様，ネットワーク輻輳率の低減を確認できた．また，全体最適化方式と局所分散方式の差がほぼ 5 %から 10 %程度の差のまま変わらない．すなわちネットワーク規模に関わらず，局所分散方式は全体最適化方式とほぼ変わらない性能を持つことが言える．

また，全体最適化方式と比べて局所分散方式がどれだけ計算時間が短縮できているか比較した．

表 5.9: ランダムトポロジの計算時間

Network type	全体最適化 [sec]	局所分散 [sec]
N 10	0.54	0.38
N 20	23.20	19.42
N 30	273.13	168.03
N 40	1514.89	869.01

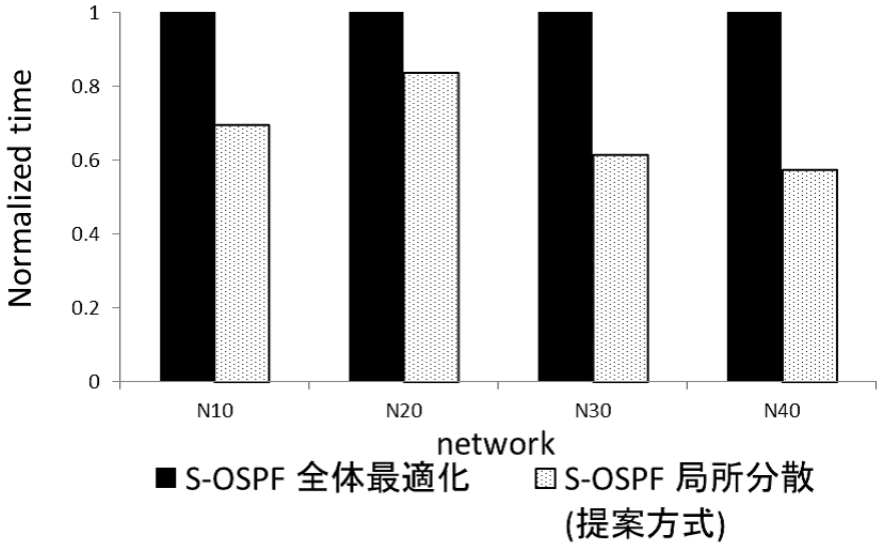

図 5.8: ランダムトポロジの計算時間の比較

結果より，ネットワーク規模が大きくなるにつれて，全体最適化方式及び局所最適化方式ともに計算時間が大きくなることが分かる．特に全体最適化方式においては大きくなる幅が顕著である．これは再計算を行う決定変数が多いのが原因である．それに対し局所分散方式は，再計算を行う決定変数がリンク故障を経由する発着ノードのみに限定されるためで，ノード数が多くなると，故障が発生したリンクを通らない発着ノードが多くなるためであると考えられる．

表 5.10: ホースモデルにおけるランダムトポロジの計算時間

Network type	全体最適化 [sec]	局所分散 [sec]
N 10	6.43	1.16
N 20	1369.71	36.85
N 30	26967.02	499.43
N 40	883401.41	42285.87

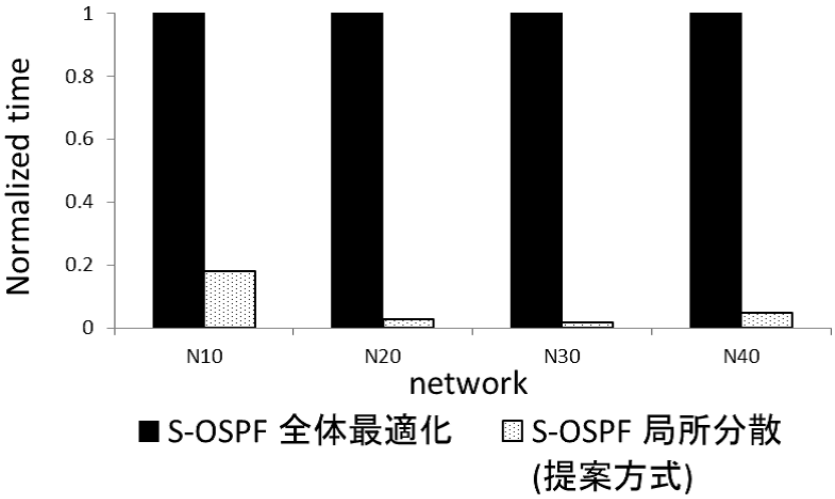

図 5.9: ホースモデルにおけるランダムトポロジの計算時間の比較

結果より，両方式ともにパイプモデルよりもさらに計算時間がかかることが分かる．これはホースモデルでは想定されうるすべてのトラヒック需要を考え，計算をしているためである．

第6章　転送システムの設計・実装

S-OSPF の分散動作を実機で確認するため，S-OPSF ネットワークの実装を行った．図 6.1 に実装アーキテクチャを示す．実装は制御対象ネットワーク (S-OSPF ネットワーク) と S-OSPF 経路計算サーバの 2 によって構成される．ノードはエッジノードと中継ノードに分けられる．中継ノードは OSPF に基づいたルーチングテーブルを持ち，ルーチングテーブルに基づいてパケットを次ノードに転送する．エッジノードは外部ネットワークと S-OSPF ネットワークの境界に存在するノードであり，発ノードの機能と中継ノードの機能の 2 つを持つ．発ノードとしての機能は外部ネットワークからパケットを受け取り，そのパケットを S-OSPF ネットワークの隣接ノードに転送することである．中継ノードとしての機能は他の中継ノードと同様である．エッジノードでは表 6.1 のようなルーチングテーブルを持つ．表 6.1(a) は S-OSPF 経路計算サーバにより設定され，表 6.1(b) は OSPF プロトコルによって決定される．

表 6.1: エッジノードにおけるルーチングテーブル

(a) For ingress function

Destination Address	Next hop	Distribution ratio
130.60.225.0/24	A	0.5
	B	0.2
	C	0.3
148.32.0.0/16	B	0.8
	C	0.2
148.32.96.0/24	A	0.3
	B	0.3
	C	0.4
⋮	⋮	⋮

(b) For transit function

Destination Address	Next hop
130.60.225.0/24	A
148.32.0.0/16	B
148.32.96.0/24	C

経路計算サーバはOSPFモジュール，SNMPモジュール，経路計算モジュール，ルータインタフェースモジュールの4つのモジュールで構成される．以下のそれぞれの機能を述べる．OSPFモジュールとSNMPモジュールはS-OSPFによる経路を計算するために必要なパラメータを収集する．OSPFモジュールはトポロジ情報やOSPFの経路情報といったルータ情報をOSPFを通じて収集する．SNMPモジュールは，各ノードにおけるインタフェースのパケット数を収集する．パケット数より，ホースモデルにおける流入/流出トラヒックの上限を決定する．経路計算モジュールでは，収集されたOSPFによるルーチング情報，link state advertisement (LSA) から計算に用いる現在のネットワークのトポロジ情報とルーチングテーブルを生成する．またSNMPによって収集されたパケット数より，ホースモデルにおけるα_p，β_qを生成する．この生成した情報に基づきS-OSPFの経路計算を行い，各エッジノードにおける発ノード用ルーチングテーブルを生成する．そして，生成されたルーチングテーブルはルータインタフェースモジュールにより，各エッジノードへ設定される．本研究では，経路計算で動作するソフトウェアを設計し，ルーチング情報とパケット数を直接入力して経路計算を行い，その結果を直接ルータにルーチングテーブルを書き加えることで動作させた．

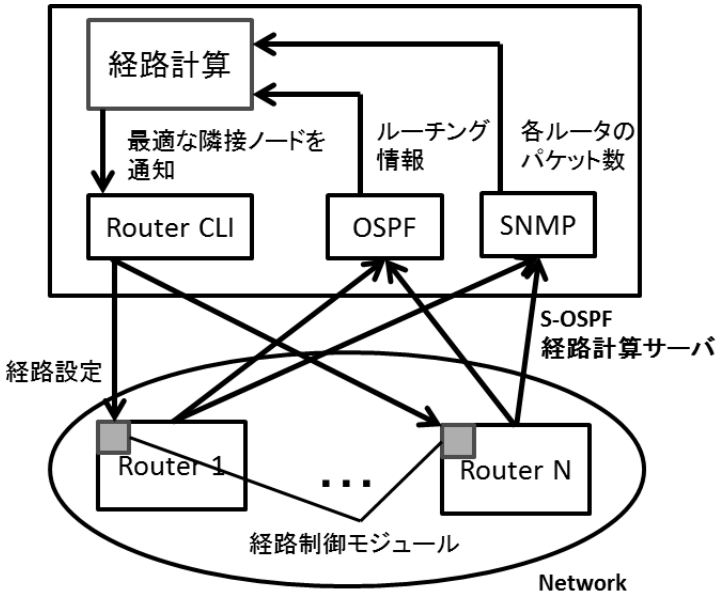

図 6.1: 転送システムの設計・実装

第7章　まとめ

本論文では，まずリンク故障においてS-OSPFの問題点をあげ，リンク故障が発生しても機能する負荷分散ルーチングの必要性を述べた．そこでS-OSPFリンク故障時におけるS-OSPFの方式として，すべてのリンクの分散比を再計算して割り当てを行う全体最適化方式に対して，リンク故障を経由する発着ノードの組にのみ再計算して割り当てを行う局所分散方式を提案した．トラヒックモデルとしてトラヒック行列が既知であるパイプモデルと，トラヒック行列が既知でないホースモデルを用いた．ネットワークトポロジとして与えられたサンプルネットワークと，ネットワークトポロジジェネレータで作成したランダムトポロジを用いてその性能評価を行った．局所分散方式は通常のOSPFと比較して，ネットワーク輻輳率を低減でき，全体最適化方式とほぼ同等の性能を持つことを示した．また，計算時間においても比較し，全体最適化方式より計算時間を短縮することができることを示した．ホースモデルにおいてもネットワーク輻輳率の低減はあまり変化がなかったが，計算時間に対しては全体最適化方式よりにも大幅に短縮することが出来た．実装においては，実装の概要を示し，S-OSPF経路計算サーバによるS-OSPFの実装方式を示した．また，Linuxベースのルータによりプロトタイプを作成し，S-OSPFの動作を確認した．現在は単一リンク故障のみを考慮しており，複数のリンクが故障した場合を想定していない．よって，今後の展望として複数リンクの故障について考える必要がある．

謝　　辞

本研究を進めるにあたり, ご指導を頂きました電気通信大学，情報理工学研究科，情報・通信工学専攻の大木英司教授に心より感謝を申し上げます. また, 研究を進めるにあたり, 様々なご指導を頂きました先輩方, また同輩の方々に厚く感謝を申し上げます.

参考文献

[1] Y. Wang and Z. Wang, "Explicit routing algorithms for internet traffic engineering," IEEE International Conference on Conputer Communications and Networks (ICCCN), 1999.

[2] B. Fortz and M. Thorup," Optimizing OSPF/IS-IS weights in a chnging would," IEEE Journal on Selected Areas in Communications, vol.20, no.4, pp. 756-767, 2002.

[3] A.K. Mishra and A. Sahoo, "S-OSPF: a traffic engineeringsolution for OSPF based on best effrot networks" IEEE Globecom 2007, pp. 1845-1849, 2007.

[4] E. Oki and A. Iwaki,"Load-Balanced IP Routing Scheme Basede on Shortest Paths in Hose Model," IEEE Trans. Commun., vol.58, no.7, pp.2088-2096, Jul. 2009.

[5] J. Chu and C. Lea, "Optimal link weights for maximizing QoS traffic," IEEE ICC 2007, pp. 610-615, 2007.

[6] M. Antic and A. Smiljanic, "Oblivious routing scheme using load balancing over shortest paths," IEEE ICC 2008, 2008.

[7] S. Tsunoda, A. H. A. Muktadir, E. Oki, "Load-Balanced Shortest-Path-Based Routing Without Traffic Splitting in Hose Model," IEEE ICC, June, 2011.

[8] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., Chuah, C.N., Ganjali, Y., and Diot, C.: "Characterization of Failures in an Operational IP Backbone Network". IEEE/ACM Trans. on Net., vol.16, No. 4, pp. 749-762, Aug. 2008.

[9] http://www.ilog.com/, 2014.

[10] "BRITE: Boston University Representative Internet Topology Generator," 2014 http://www.cs.bu.edu/brite/

[11] I.M. Kamrul and E. Oki, " PSO: Preventive Start-Time Optimization of OSPF Link Weights to Counter Network Failure, "IEEE Commun. Letters, vol. 14, no. 6, Jun. 2010.

研究実績

査読付き論文

M. Honma, S. Tsunoda, and E. Oki, ”Load-Balanced Routing with Selective Even Traffic Splitting” Progress in Informatics, No.10, March. 2013.

本間 奨，大木 英司, “ リンク故障を考慮した IP 負荷分散ルーチング方式, ”電子情報通信学会論文誌（投稿予定）

外部発表

M. Honma and E. Oki, ”Load-Balanced Shortest-Path Based Routing with Even Traffic Splitting,” The 18th Asia-Pacific Conference on Communications (APCC2012), Jeju island, Korea, Oct. 2012.

M. Honma and E.Oki, ”Performance of Smart-OSPF with Even Traffic Splitting,” International Workshop on Modern Science and Technology (IWMST2012), Aug. 2012.

本間 奨，大木 英司, ”S-OSPF における均等分散ルーチング方式,” 2012 年 3 月電子情報通信学会ネットワークシステム研究会電子情報通信学会技術研究報告，vol. 111, no. 468, NS2011-228, pp. 275-278, 2012 年 3 月.

本間 奨，大木 英司, ”リンク故障を考慮した IP 負荷分散ルーチング方式,” IEICE, PN 研究会 IEICE 技術研究報告, 2014 年 3 月.

発表予定

本間 奨，大木 英司, ”リンク故障を考慮した IP 負荷分散ルーチング方式,” 2014 年 3 月電子情報通信学会フォトニックネットワーク研究会電子情報通信学会技術研究報告 2014 年 3 月.

付録

プログラムリスト

初期設定とすべてのネットワーク輻輳率の出力プログラム

```
1   #include "congestion_calc.h"
2
3   double getrusage_sec()
4   {
5     struct rusage t;
6     struct timeval tv;
7     getrusage(RUSAGE_SELF, &t);
8     tv = t.ru_utime;
9     return tv.tv_sec + tv.tv_usec * 1e-6;
10  }
11
12  int main(int argc, char *argv[])
13  {
14    unsigned int  node_num;
15    unsigned long rseed = (unsigned long)time(NULL);
16
17    unsigned int   **adjacencymatrix;
18    unsigned int   **capacitymatrix;
19    unsigned int   **trafficmatrix;
20    double         **congestmatrix;
21
22    unsigned int Cmax = 50;
23    unsigned int Imax = 50;
24    unsigned long tabu_num = 1000000;
25
26    int          ch;
27    extern char *optarg;
28    extern int   optind, opterr;
29
30    char filename[256];
31    /* char *capafile; */
32    /* char *traffile; */
33
34    int traffic_base, traffic_amp, capacity_base, capacity_amp;
35    int i, j, k;
36
37    char *token;
38
39    double congestion, start, end, cong, Worst_cong;
40
41    /* memory check */
42    /* unsigned long long memory; */
43
44    traffic_base=0, traffic_amp=1000, capacity_base=8000, capacity_amp=4000;
45
```

```
46
47      while ((ch = getopt(argc, argv, "r:t:c:hi:d:")) != -1){
48        switch (ch){
49        case 'r':
50          rseed = atol(optarg);
51          break;
52        case 'd':
53          token        = strtok(optarg, ":");
54          traffic_base = atoi(token);
55          token        = strtok(NULL, ":");
56          traffic_amp  = atoi(token) - traffic_base;
57          break;
58        case 'c':
59
60          if (strchr(optarg, ':') == NULL){
61            puts("Try `./tabu_search -h' for more information.");
62            exit(1);
63            break;
64          }
65          token         = strtok(optarg, ":");
66          capacity_base = atoi(token);
67          token         = strtok(NULL, ":");
68          capacity_amp  = atoi(token) - capacity_base;
69          break;
70        case 'i':
71
72          if (strchr(optarg, ':') == NULL){
73            puts("Try `./tabu_search -h' for more information.");
74            exit(1);
75            break;
76          }
77          token = strtok(optarg, ":");
78          Cmax  = atoi(token);
79          token = strtok(NULL, ":");
80          Imax  = atoi(token);
81          break;
82        case 'h':
83          usage();
84          exit(0);
85        case 't':
86          tabu_num = atol(optarg);
87          break;
88        case ':':
89          puts("Try `./tabu_search -h' for more information.");
90          exit(1);
91          break;
92        case '?':
93          puts("Try `./tabu_search -h' for more information.");
94          exit(1);
95          break;
96        }
97      }
98      argc -= optind;
99      argv += optind;
100
101     /* start = getrusage_sec();  */
102
103     if (argc != 1) {
104       usage();
105       exit(1);
```

```
106     }
107     strcpy(filename, argv[0]);
108
109     /* 簸掩違   */
110     srand48(rseed);
111
112     /* printf("input file: %s\n", filename); */
113     if((node_num = read_node_num(filename)) <= 0){
114       fprintf (stderr, "read error on node number!\n");
115       exit(1);
116     }
117
118     if((adjacencymatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
119         == NULL) MEM_EXIT;
120     if((capacitymatrix  = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
121         == NULL) MEM_EXIT;
122     if((trafficmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
123         == NULL) MEM_EXIT;
124     if((congestmatrix   = (double **)malloc(node_num * sizeof(double *)))       == N
125   ULL) MEM_EXIT;
126     for (i=0; i < node_num; ++i){
127       if((adjacencymatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
128        == NULL) MEM_EXIT;
129       memset(adjacencymatrix[i], 0, node_num);
130       if((capacitymatrix[i]  = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
131        == NULL) MEM_EXIT;
132       memset(capacitymatrix[i], 0, node_num);
133       if((trafficmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
134        == NULL) MEM_EXIT;
135       memset(trafficmatrix[i], 0, node_num);
136       if((congestmatrix[i] = (double *)malloc(node_num * sizeof(double)))     == NULL)
137    MEM_EXIT;
138       memset(congestmatrix[i], 0, node_num);
139     }
140
141     make_adjacency_matrix(filename, adjacencymatrix);
142
143     /* make_capacity_matrix(capacitymatrix); */
144     /* make_traffic_matrix(trafficmatrix); */
145
146     make_random_capacity_matrix02(node_num, adjacencymatrix, capacitymatrix,
147    capacity_base, capacity_amp);
148     make_random_traffic_matrix(node_num, trafficmatrix, traffic_base, traffic_amp);
149
150     /* printf("Capacity i, j:\n"); */
151     /* print_uint_matrix02 (capacitymatrix,node_num,node_num); */
152     /* printf("Traffic i, j:\n"); */
153     /* print_uint_matrix02 (trafficmatrix,node_num,node_num); */
154
155     for(i=0; i<node_num; ++i){
156       for(j=0; j<node_num; ++j){
157         congestmatrix[i][j] = 1000;
158       }
159     }
160
161     cong = 0;
162     Worst_cong = 0;
163     k = 0;
164
165     congestion = spr_pipe_congestion(node_num, adjacencymatrix, capacitymatrix, trafficmatrix);
```

```
166   printf("SPR_pipe : %.20f\n", congestion);
167   /* printf("%.20f ", congestion); */
168
169   congestion = spr_hose_congestion(node_num, adjacencymatrix, capacitymatrix, trafficmatrix);
170   printf("SPR_hose : %.20f\n", congestion);
171   /* /\* printf("%.20f ", congestion); *\/ */
172
173   if ((congestion = sospf_split_pipe_congestion(node_num, adjacencymatrix, capacitymatrix,
174  trafficmatrix)) < 0){
175     fprintf(stderr, "error on calculate sospf\n");
176     goto TERMINATE;
177   }
178   printf("SOSPF-S : %.20f\n", congestion);
179   /* printf("%.20f ", congestion); */
180
181   if ((congestion = sospf_split_hose_congestion(node_num, adjacencymatrix, capacitymatrix,
182 trafficmatrix)) < 0){
183     fprintf(stderr, "error on calculate sospf_hose\n");
184     /* goto TERMINATE; */
185   }
186   /* printf("SOSPF-S_hose : %.20f\n", congestion); */
187   /* printf("%.20f ", congestion); */
188
189
190   spr_pipe_congestion_lf(node_num, adjacencymatrix, capacitymatrix, trafficmatrix,
191 congestmatrix);
192   for (i=0; i < node_num; ++i){
193     for (j=0; j < node_num; ++j){
194       if(i < j && congestmatrix[i][j] != 1000){
195         printf("SPR_LF_pipe_link(%d,%d) : %.20f\n", i, j, congestmatrix[i][j]);
196         /* printf("%.20f ", congestmatrix[i][j]); */
197         if (congestmatrix[i][j] <= 100){
198           cong += congestmatrix[i][j];
199           k++;
200           if(Worst_cong < congestmatrix[i][j])
201             Worst_cong = congestmatrix[i][j];
202         }
203       }
204     }
205   }
206   /* printf("k %d\n", k); */
207   cong = cong / k;
208   printf("SPR_LF_pipe_Ave : %.20f\n", cong);
209   cong = 0;
210   printf("SPR_LF_pipe_Wor : %.20f\n\n", Worst_cong);
211   Worst_cong = 0;
212   k = 0;
213
214
215
216   sospf_pipe_congestion_all_reroute(node_num, adjacencymatrix, capacitymatrix,
217  trafficmatrix, congestmatrix);
218   for (i=0; i < node_num; ++i){
219     for (j=i+1; j < node_num; ++j){
220       if(congestmatrix[i][j] != 1000){
221         printf("SOSPF-S_All_re_pipe(%d,%d)  : %.20f\n", i, j, congestmatrix[i][j]);
222         /* printf("%.20f ", congestmatrix[i][j]); */
223         if (congestmatrix[i][j] <= 100){
224           cong += congestmatrix[i][j];
225           k++;
```

```
226             if(Worst_cong < congestmatrix[i][j])
227               Worst_cong = congestmatrix[i][j];
228           }
229         }
230       }
231     }
232     cong = cong / k;
233     printf("SOSPF-S_All_re_pipe_Ave  : %.20f\n", cong);
234     cong = 0;
235     printf("SOSPF-S_All_re_pipe_Wor  : %.20f\n\n", Worst_cong);
236     Worst_cong = 0;
237     k = 0;
238
239     sospf_pipe_local_reroute(node_num, adjacencymatrix, capacitymatrix, trafficmatrix,
240    congestmatrix);
241     for (i=0; i < node_num; ++i){
242       for (j=0; j < node_num; ++j){
243         if(i < j && congestmatrix[i][j] != 1000){
244           printf("SOSPF-S_Local_re_pipe_link(%d,%d) : %.20f\n", i, j, congestmatrix[i][j]);
245           /* printf("%.20f ", congestmatrix[i][j]); */
246           if (congestmatrix[i][j] <= 100){
247             cong += congestmatrix[i][j];
248             k++;
249             if(Worst_cong < congestmatrix[i][j])
250               Worst_cong = congestmatrix[i][j];
251           }
252         }
253       }
254     }
255
256     cong = cong / k;
257     printf("SOSPF-S_local_reroute_p_Ave : %.20f\n", cong);
258     cong = 0;
259     printf("SOSPF-S_local_reroute_p_Wor : %.20f\n\n", Worst_cong);
260     Worst_cong = 0;
261     k = 0;
262
263     spr_hose_congestion_lf(node_num, adjacencymatrix, capacitymatrix, trafficmatrix,
264    congestmatrix);
265     for (i=0; i < node_num; ++i){
266       for (j=0; j < node_num; ++j){
267         if(i < j && congestmatrix[i][j] != 1000){
268           printf("SPR_LF_hose_link(%d,%d)  : %.20f\n", i, j, congestmatrix[i][j]);
269           /* printf("%.20f ", congestmatrix[i][j]); */
270           if (congestmatrix[i][j] <= 100){
271             cong += congestmatrix[i][j];
272             k++;
273             if(Worst_cong < congestmatrix[i][j])
274               Worst_cong = congestmatrix[i][j];
275           }
276         }
277       }
278     }
279     cong = cong / k;
280     printf("SPR_LF_hose_Ave  : %.20f\n", cong);
281     cong = 0;
282     printf("SPR_LF_hose_Wor : %.20f\n\n", Worst_cong);
283     Worst_cong = 0;
284     k = 0;
285
```

```
286     /* SOSPF-S_All_reroute_hose 荐膊鏢 routing_scheme.c T */
287     sospf_hose_congestion_all_reroute(node_num, adjacencymatrix, capacitymatrix, trafficmatrix,
288   congestmatrix);
289     for (i=0; i < node_num; ++i){
290       for (j=i+1; j < node_num; ++j){
291         if(congestmatrix[i][j] != 1000){
292           printf("SOSPF-S_All_re_hose(%d,%d)  : %.20f\n", i, j, congestmatrix[i][j]);
293           /* printf("%.20f ", congestmatrix[i][j]); */
294           if (congestmatrix[i][j] <= 100){
295             cong += congestmatrix[i][j];
296             k++;
297             if(Worst_cong < congestmatrix[i][j])
298               Worst_cong = congestmatrix[i][j];
299           }
300         }
301       }
302     }
303     cong = cong / k;
304     printf("SOSPF-S_All_re_hose_Ave  : %.20f\n", cong);
305     cong = 0;
306     printf("SOSPF-S_All_re_hose_Wor  : %.20f\n\n", Worst_cong);
307     Worst_cong = 0;
308     k = 0;
309
310   /* SOSPF-S_Local_reroute_hose 荐膊鏢 routing_scheme.c T */
311     sospf_hose_local_reroute(node_num, adjacencymatrix, capacitymatrix, trafficmatrix, congestmatrix
312     for (i=0; i < node_num; ++i){
313       for (j=0; j < node_num; ++j){
314         if(i < j && congestmatrix[i][j] != 1000){
315           printf("SOSPF-S_Local_re_hose_link(%d,%d) : %.20f\n", i, j, congestmatrix[i][j]);
316           /* printf("%.20f ", congestmatrix[i][j]); */
317           if (congestmatrix[i][j] <= 100){
318             cong += congestmatrix[i][j];
319             k++;
320             if(Worst_cong < congestmatrix[i][j])
321               Worst_cong = congestmatrix[i][j];
322           }
323         }
324       }
325     }
326
327     cong = cong / k;
328     printf("SOSPF-S_local_reroute_h_Ave : %.20f\n", cong);
329     cong = 0;
330     printf("SOSPF-S_local_reroute_h_Wor : %.20f\n\n", Worst_cong);
331     Worst_cong = 0;
332     k = 0;
333
334     end = getrusage_sec();
335     printf("%f 腱障\n",(end-start));
336
337     putchar('\n');
338   TERMINATE:
339
340     for (i=0; i < node_num; ++i){
341       free(adjacencymatrix[i]);
342       adjacencymatrix[i] = NULL;
343       free(capacitymatrix[i]);
344       capacitymatrix[i] = NULL;
345       free(trafficmatrix[i]);
```

```
346       trafficmatrix[i] = NULL;
347       free(congestmatrix[i]);
348       congestmatrix[i] = NULL;
349     }
350     free(adjacencymatrix);
351     adjacencymatrix = NULL;
352     free(capacitymatrix);
353     capacitymatrix = NULL;
354     free(trafficmatrix);
355     trafficmatrix = NULL;
356     free(congestmatrix);
357     congestmatrix = NULL;
358 
359     return 0;
360   }
```

パラメータの設定プログラム

```
#include "matrix.h"

int
make_adjacency_matrix (char          *file,
                       unsigned int **adjacencymatrix)
{
  FILE *fp = fopen(file, "r");
  char buf[350];
  char *token;
  int source, dest;

  memset(buf, 0, sizeof(buf)/sizeof(buf[0]));

  while(fgets(buf, sizeof(buf), fp) != NULL) {
    switch (buf[0]) {
    case '#': break;
    case 'N': break;
    default:
      token  = strtok(buf, ": ");
      source = atoi(token);
      while((token = strtok(NULL, ": "))){
        dest = atoi(token);
        adjacencymatrix[source][dest] = 1;
      }
      break;
    }
  }
  return 0;
}

int
make_capacity_matrix (unsigned int **capacitymatrix)
{
  FILE *fp;
  char buf[350];
  char *token;
  int i, j;

  fp = fopen("Capa.txt", "r");
  memset(buf, 0, sizeof(buf)/sizeof(buf[0]));

  while(fgets(buf, sizeof(buf), fp) != NULL) {
    switch (buf[0]) {
    case '#': break;
    case 'N': break;
    default:
      token  = strtok(buf, " ");
      i = atoi(token);
      token  = strtok(NULL, ": ");
      j = atoi(token);
      while((token = strtok(NULL, ": "))){
        capacitymatrix[i][j] = atoi(token);
      }
      break;
    }
  }
  fclose(fp);
  return 0;
```

```
59  }
60
61  int
62  make_traffic_matrix (unsigned int **trafficmatrix)
63  {
64    FILE *fp;
65    char buf[350];
66    char *token;
67    int i, j;
68
69    fp = fopen("Traf.txt", "r");
70    memset(buf, 0, sizeof(buf)/sizeof(buf[0]));
71
72    while(fgets(buf, sizeof(buf), fp) != NULL) {
73      switch (buf[0]) {
74      case '#': break;
75      case 'N': break;
76      default:
77        token  = strtok(buf, " ");
78        i = atoi(token);
79        token  = strtok(NULL, ": ");
80        j = atoi(token);
81        while((token = strtok(NULL, ": "))){
82          trafficmatrix[i][j] = atoi(token);
83        }
84        break;
85      }
86    }
87    fclose(fp);
88    return 0;
89  }
90
91  int
92  make_random_matrix (int            node_num,
93                      unsigned int **adjacencymatrix,
94                      unsigned int **matrix,
95                      int            base,
96                      int            amp)
97  {
98    int i,j;
99    for (i=0; i<node_num; ++i){
100     for (j=0; j<node_num; ++j){
101       if(adjacencymatrix[i][j] == 0){
102         matrix[i][j] = 0;
103       }else{
104         matrix[i][j] = base + (amp * drand48());
105       }
106     }
107   }
108   return 0;
109 }
110
111 int
112 make_linkcost_matrix_based_capacity (int            node_num,
113                                      unsigned int **capacitymatrix,
114                                      unsigned int **matrix)
115 {
116   int i,j;
117   for (i=0; i<node_num; ++i){
118     for (j=0; j<node_num; ++j){
```

```
119         if(capacitymatrix[i][j] == 0){
120           matrix[i][j] = 1000;
121         }else{
122           matrix[i][j] = (unsigned int)(1.0/(double)capacitymatrix[i][j]*65536.0);
123         }
124       }
125     }
126     return 0;
127   }
128
129
130   int
131   make_random_capacity_matrix (int            node_num,
132                                unsigned int **adjacencymatrix,
133                                unsigned int **matrix,
134                                int            base,
135                                int            amp)
136   {
137     int i,j;
138     for (i=0; i<node_num; ++i){
139       for (j=0; j<node_num; ++j){
140         if(adjacencymatrix[i][j] == 0){
141           matrix[i][j] = 0;
142         }else{
143           matrix[i][j] = base + (amp * drand48());
144         }
145       }
146     }
147     return 0;
148   }
149
150   int
151   make_random_capacity_matrix02 (int            node_num,
152                                  unsigned int **adjacencymatrix,
153                                  unsigned int **matrix,
154                                  int            base,
155                                  int            amp)
156   {
157     int i,j;
158     for (i=0; i<node_num; ++i){
159       for (j=i+1; j<node_num; ++j){
160         if(adjacencymatrix[i][j] == 0){
161           matrix[i][j] = 0;
162         }else{
163           matrix[i][j] = base + (amp * drand48());
164           matrix[j][i] = matrix[i][j];
165         }
166       }
167     }
168     return 0;
169   }
170
171   int
172   make_random_traffic_matrix (int            node_num,
173                               unsigned int **matrix,
174                               int            base,
175                               int            amp)
176   {
177     int i,j;
178     for (i=0; i<node_num; ++i){
```

```
179       for (j=0; j<node_num; ++j){
180         if(i == j){
181           matrix[i][j] = 0;
182         }else{
183           matrix[i][j] = base + (amp * drand48());
184         }
185       }
186     }
187     return 0;
188   }
189
190   int
191   print_uint_matrix (unsigned int **matrix,
192                      int            size_x,
193                      int            size_y)
194   {
195     int i,j;
196
197     for (i=0; i < size_x; ++i){
198       for (j=0; j < size_y; ++j){
199         printf ("%10d ", matrix[i][j]);
200       }
201       putchar('\n');
202     }
203     return 0;
204   }
205
206   int
207   print_uchar_matrix (unsigned char **matrix,
208                       int            size_x,
209                       int            size_y)
210   {
211     int i,j;
212
213     for (i=0; i < size_x; ++i){
214       for (j=0; j < size_y; ++j){
215         printf ("%10d ", matrix[i][j]);
216       }
217       putchar('\n');
218     }
219     return 0;
220   }
221
222   int
223   print_double_matrix (double **matrix,
224                        int      size_x,
225                        int      size_y)
226   {
227     int i,j;
228
229     for (i=0; i < size_x; ++i){
230       for (j=0; j < size_y; ++j){
231         printf ("%10.6f ", matrix[i][j]);
232       }
233       putchar('\n');
234     }
235     return 0;
236   }
237
238   int
```

```
print_ancestor_matrix (unsigned int ***matrix,
                       int             size_x,
                       int             size_y,
                       int             size_z)
{
  int i,j,k;
  for (i=0; i < size_x; ++i){
    for (j=0; j < size_y; ++j){
      for (k=0; k < size_z; ++k){
        printf ("dest:%d source:%d current:%d value:%d\n", i, j, k, matrix[i][j][k]);
      }
    }
  }
  return 0;
}

int
double_matrix_cmp (int      num,
                   double **matrix1,
                   double ** matrix2)
{
  int i,j;

  for (i=0; i<num; ++i){
    for (j=0; j<num; ++j){
      if (matrix1[i][j] != matrix2[i][j]){
        return (-1);
      }
    }
  }
  return (1);
}

int
uint_matrix_cmp (int            num,
                 unsigned int **matrix1,
                 unsigned int **matrix2)
{

  int i,j;

  for (i=0; i<num; ++i){
    for (j=0; j<num; ++j){
      if (matrix1[i][j] != matrix2[i][j]){
        return (-1);
      }
    }
  }
  return (1);
}

void
double_matrix_zeroset (int num,
                       double **matrix)
{
  int i,j;

  for (i=0; i<num; ++i){
    for (j=0; j<num; ++j){
      matrix[i][j] = 0;
```

```
299       }
300     }
301   }
302
303   void
304   double_matrix_cpy (int num,
305                      double **matrix1,
306                      double **matrix2)
307   {
308     int i,j;
309
310     for (i=0; i<num; ++i){
311       for (j=0; j<num; ++j){
312         matrix1[i][j] = matrix2[i][j];
313       }
314     }
315   }
316
317   int
318   print_uint_matrix02 (unsigned int **matrix,
319                        int           size_x,
320                        int           size_y)
321
322   {
323     int i,j;
324
325     for (i=0; i < size_x; ++i){
326       for (j=0; j < size_y; ++j){
327         printf ("%d %d:%d\n", i, j, matrix[i][j]);
328       }
329       /* putchar('\n'); */
330     }
331     return 0;
332   }
333
334   int
335   print_double_matrix02 (double **matrix,
336                          int           size_x,
337                          int           size_y)
338   {
339     int i,j;
340
341     for (i=0; i < size_x; ++i){
342       for (j=0; j < size_y; ++j){
343         printf ("%d %d:%10.6f\n", i, j, matrix[i][j]);
344       }
345       /* putchar('\n'); */
346     }
347     return 0;
348   }
349
350   void
351   division_matrix_cpy (int num,
352                unsigned int **matrix1,
353                unsigned int **matrix2)
354   {
355     int i,j;
356
357     for (i=0; i<num; ++i){
358       for (j=0; j<num; ++j){
```

```
359         matrix1[i][j] = matrix2[i][j];
360       }
361     }
362   }
363
364   int
365   print_test (double  ****matrix,
366               int         node_num)
367   {
368     int p,q,i,j;
369
370     for (p=0; p < node_num; ++p){
371       for (q=0; q < node_num; ++q){
372         if(p != q){
373           printf ("p=%d, q=%d\n", p, q);
374           for (i=0; i < node_num; ++i){
375             for (j=0; j < node_num; ++j){
376               if(matrix[p][q][i][j] != 0){
377                 printf ("x[p][q][%d][%d]:%10.6f\n", i, j, matrix[p][q][i][j]);
378               }
379             }
380             /* putchar('\n'); */
381           }
382         }
383       }
384     }
385     putchar('\n');
386     return 0;
387   }
388
389   int
390   cal_break_num (double  ****matrix,
391                  int         node_num,
392                  int         b_i,
393                  int         b_j)
394
395   {
396     int p,q,i,j,num;
397
398     num = 0;
399     for (p=0; p<node_num; ++p){
400       for (q=0; q<node_num; ++q){
401         if((matrix[p][q][b_i][b_j]==0)&&(matrix[p][q][b_j][b_i]==0)){
402           for (i=0; i<node_num; ++i){
403             if(i==p){
404               for (j=0; j<node_num; ++j){
405                 num += 1;
406               }
407             }
408           }
409         }
410       }
411     }
412     return num;
413   }
```

```
#include "optimize.h"

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */

int
opt_sospf_split_pipe (const int        node_num,
                      unsigned int   **nexthopmatrix,
                      unsigned int  ***ancestormatrix,
                      unsigned int   **capacitymatrix,
                      unsigned int   **trafficmatrix,
                      double       ****routingmatrix)
{

  unsigned long col_num     = node_num * node_num * node_num * node_num + 1;
  unsigned long row_num     =
    node_num * (node_num-1) * (node_num-2) +
    node_num * (node_num-1) +
    (node_num * node_num);
  unsigned long nonzero_num =
    (node_num+1)*(node_num*(node_num-1)*(node_num-2)) +
    (2*(node_num-1))*(node_num*(node_num-1)) +
    (node_num*node_num+1)*(node_num*node_num);

  double *rhs; //right hand side
  double *lb; //lower bound
  double *ub; //upper bound
  double *obj;
  char   *sense;
  char   **rowname;
  char   **colname;
  int    *rowlist;
  int    *collist;
  double *vallist;

  int i,j,p,q;
  unsigned long ptr = 0;
  unsigned long row = 0;

  /* double  objval, maxviol; */
  int     cur_numrows, cur_numcols;
  double *x     = NULL;
  int    *cstat = NULL;
  int    *rstat = NULL;

  CPXENVptr env    = NULL;
  CPXLPptr  lp     = NULL;
  int       status = 0;
  int       solnstat, solnmethod, solntype;

  /* char *basismsg; */

```

```
59
60      if ((rhs = (double *)malloc(row_num * sizeof(double))) == NULL){
61        status = -1;
62        fprintf(stderr, "Memory allocate error\n");
63        goto TERMINATE;
64      }
65
66      if ((lb = (double *)malloc(col_num * sizeof(double))) == NULL){
67        status = -1;
68        fprintf(stderr, "Memory allocate error\n");
69        goto TERMINATE;
70      }
71
72      if ((ub = (double *)malloc(col_num * sizeof(double))) == NULL){
73        status = -1;
74        fprintf(stderr, "Memory allocate error\n");
75        goto TERMINATE;
76      }
77
78      if ((obj = (double *)malloc(col_num * sizeof(double))) == NULL){
79        status = -1;
80        fprintf(stderr, "Memory allocate error\n");
81        goto TERMINATE;
82      }
83
84      if ((sense = (char *)malloc(row_num * sizeof(char))) == NULL){
85        status = -1;
86        fprintf(stderr, "Memory allocate error\n");
87        goto TERMINATE;
88      }
89
90      if ((rowlist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
91        status = -1;
92        fprintf(stderr, "Memory allocate error\n");
93        goto TERMINATE;
94      }
95
96      if ((collist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
97        status = -1;
98        fprintf(stderr, "Memory allocate error\n");
99        goto TERMINATE;
100     }
101
102     if ((vallist = (double *)malloc(nonzero_num * sizeof(double))) == NULL){
103       status = -1;
104       fprintf(stderr, "Memory allocate error\n");
105       goto TERMINATE;
106     }
107
108     if ((rowname = (char **)malloc(row_num * sizeof(char *))) == NULL){
109       status = -1;
110       fprintf(stderr, "Memory allocate error\n");
111       goto TERMINATE;
112     }
113     for (i=0; i<row_num; ++i){
114       if ((rowname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
115         status = -1;
116         fprintf(stderr, "Memory allocate error\n");
117         goto TERMINATE;
118       }
```

```
119     }
120
121     if ((colname = (char **)malloc(col_num * sizeof(char *))) == NULL){
122       status = -1;
123       fprintf(stderr, "Memory allocate error\n");
124       goto TERMINATE;
125     }
126     for (i=0; i<col_num; ++i){
127       if ((colname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
128         status = -1;
129         fprintf(stderr, "Memory allocate error\n");
130         goto TERMINATE;
131       }
132     }
133
134     for (i=0; i<node_num; ++i){
135       for (p=0; p<node_num; ++p){
136         for (q=0; q<node_num; ++q){
137           if ((i!=p)&&(i!=q)&&(p!=q)){
138             sprintf(rowname[row], "INTERNAL(%d,%d,%d)", i, p, q);
139             sense[row] = 'E';
140             rhs[row]   = 0.0;
141             row++;
142           }
143         }
144       }
145     }
146
147
148     for (p=0; p<node_num; ++p){
149       for (q=0; q<node_num; ++q){
150         if ((p!=q)){
151           sprintf(rowname[row], "SOURCE(%d,%d,%d)", p, p, q);
152           sense[row] = 'E';
153           rhs[row]   = 1.0;
154           row++;
155         }
156       }
157     }
158
159
160     for (p=0; p<node_num; ++p){
161       for (q=0; q<node_num; ++q){
162         sprintf(rowname[row], "CAPACITY(%d,%d)", p, q);
163         sense[row] = 'L';
164         rhs[row]   = 0.0;
165         row++;
166       }
167     }
168
169
170     ptr      = 0;
171     obj[ptr] = 1.0;
172     ub[ptr]  = CPX_INFBOUND;
173     lb[ptr]  = 0.0;
174     sprintf(colname[ptr], "r");
175     ptr++;
176     for (i=0; i<node_num; ++i){
177       for (j=0; j<node_num; ++j){
178         for (p=0; p<node_num; ++p){
```

```
        for (q=0; q<node_num; ++q){
          obj[ptr] = 0.0;
          ub[ptr]  = 1.0;
          lb[ptr]  = 0.0;
          sprintf(colname[ptr], "x(%d,%d,%d,%d)", i, j, p, q);
          ptr++;
        }
      }
    }
  }


  ptr = 0;
  row = 0;
  for (i=0; i<node_num; ++i){
    for (p=0; p<node_num; ++p){
      for (q=0; q<node_num; ++q){
        if ((i!=p)&&(i!=q)&&(p!=q)){
          rowlist[ptr] = row;
          collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
i*node_num + nexthopmatrix[q][i] + 1;
          vallist[ptr] = 1.0;
          ptr++;
          for (j = 0; j<node_num; ++j){
            rowlist[ptr] = row;
            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
j*node_num + i + 1;
            /* x(p,q,j,i) => p*n^3 + q*n^2 + j*n + i  э r+1*/
            vallist[ptr] = -1.0;
            ptr++;
          }
          row++;
        }
      }
    }
  }



  for (p=0; p<node_num; ++p){
    for (q=0; q<node_num; ++q){
      if ((p != q)){
        for (j=0; j<node_num; j++){
          if ((ancestormatrix[q][p][j] == 0) && (j != p)){
            rowlist[ptr] = row;
            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
p*node_num + j + 1;
            vallist[ptr] = 1.0;
            ptr++;
          }
        }
        for (j=0; j<node_num; j++){
          if (j != p){
            rowlist[ptr] = row;
            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
j*node_num + p + 1;
            vallist[ptr] = -1.0;
            ptr++;
          }
        }
```

```
239           row++;
240         }
241       }
242     }
243
244
245     for (i=0; i<node_num; ++i){
246       for (j=0; j<node_num; ++j){
247         rowlist[ptr] = row;
248         collist[ptr] = 0;
249         vallist[ptr] = -1.0 * capacitymatrix[i][j];
250         ptr++;
251         for (p=0; p<node_num; ++p){
252           for (q=0; q<node_num; ++q){
253             rowlist[ptr] = row;
254             collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
255     i*node_num + j + 1;
256             vallist[ptr] = trafficmatrix[p][q];
257             ptr++;
258           }
259         }
260         row++;
261       }
262     }
263
264     if (ptr > nonzero_num){
265       status = -1;
266       fprintf (stderr, "non zero number is small!");
267       goto TERMINATE;
268     }
269
270     env = CPXopenCPLEX(&status);
271
272     if (env == NULL){
273       char  errmsg[CPXMESSAGEBUFSIZE];
274       fprintf (stderr, "Could not open CPLEX environment.\n");
275       CPXgeterrorstring (env, status, errmsg);
276       fprintf (stderr, "%s", errmsg);
277       goto TERMINATE;
278     }
279
280     /* status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON); */
281     /* if ( status ) { */
282     /*   fprintf (stderr, "Failure to turn on screen indicator, error %d.\n",
283     status); */
284     /*   goto TERMINATE; */
285     /* } */
286
287     lp = CPXcreateprob (env, &status, "SOSPFSPLIT");
288     if ( lp == NULL ) {
289       fprintf (stderr, "Failed to create LP.\n");
290       goto TERMINATE;
291     }
292
293     CPXchgobjsen (env, lp, CPX_MIN);  /* Problem is minimize */
294
295     status = CPXnewrows (env, lp, row_num, rhs, sense, NULL, rowname);
296     if ( status ) {
297       fprintf (stderr, "Failed to newrows.\n");
298       goto TERMINATE;
```

```
299     }
300
301     status = CPXnewcols (env, lp, col_num, obj, lb, ub, NULL, colname);
302     if ( status ) {
303       fprintf (stderr, "Failed to newcols.\n");
304       goto TERMINATE;
305     }
306
307
308     status = CPXchgcoeflist (env, lp, ptr, rowlist, collist, vallist);
309     if ( status ) {
310       fprintf (stderr, "Failed to chgcoeflist.\n");
311       goto TERMINATE;
312     }
313
314     /* status = CPXwriteprob(env, lp, "check.lp", NULL); */
315     /* if ( status ) { */
316     /*   fprintf (stderr, "Failed to write prob.\n"); */
317     /*   goto TERMINATE; */
318     /* } */
319
320     status   = CPXlpopt(env, lp);
321     solnstat = CPXgetstat (env, lp);
322
323     if      ( solnstat == CPX_STAT_UNBOUNDED ) {
324       printf ("Model is unbounded\n");
325       goto TERMINATE;
326     }
327     else if ( solnstat == CPX_STAT_INFEASIBLE ) {
328       printf ("Model is infeasible\n");
329       goto TERMINATE;
330     }
331     else if ( solnstat == CPX_STAT_INForUNBD ) {
332       printf ("Model is infeasible or unbounded\n");
333       goto TERMINATE;
334     }
335
336     status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
337     if ( status ) {
338       fprintf (stderr, "Failed to obtain solution info.\n");
339       goto TERMINATE;
340     }
341     /* printf ("Solution status %d, solution method %d\n", solnstat, solnmethod); */
342
343     if ( solntype == CPX_NO_SOLN ) {
344       fprintf (stderr, "Solution not available.\n");
345       goto TERMINATE;
346     }
347
348     /* status = CPXgetobjval (env, lp, &objval); */
349     /* if ( status ) { */
350     /*   fprintf (stderr, "Failed to obtain objective value.\n"); */
351     /*   goto TERMINATE; */
352     /* } */
353     /* printf ("Objective value %.10g.\n", objval); */
354
355     cur_numcols = CPXgetnumcols (env, lp);
356     cur_numrows = CPXgetnumrows (env, lp);
357
358     /* Retrieve basis, if one is available */
```

```
359
360     if ( solntype == CPX_BASIC_SOLN ) {
361       cstat = (int *) malloc (cur_numcols*sizeof(int));
362       rstat = (int *) malloc (cur_numrows*sizeof(int));
363       if ( cstat == NULL || rstat == NULL ) {
364         fprintf (stderr, "No memory for basis statuses.\n");
365         goto TERMINATE;
366       }
367
368       status = CPXgetbase (env, lp, cstat, rstat);
369       if ( status ) {
370         fprintf (stderr, "Failed to get basis; error %d.\n", status);
371         goto TERMINATE;
372       }
373     }
374     else {
375       printf ("No basis available\n");
376     }
377
378     /* Retrieve solution vector */
379     x = (double *) malloc (cur_numcols*sizeof(double));
380     if ( x == NULL ) {
381       fprintf (stderr, "No memory for solution.\n");
382       goto TERMINATE;
383     }
384
385     status = CPXgetx (env, lp, x, 0, cur_numcols-1);
386     if ( status ) {
387       fprintf (stderr, "Failed to obtain primal solution.\n");
388       goto TERMINATE;
389     }
390
391     ptr = 1;
392     for (p=0; p<node_num; ++p){
393       for (q=0; q<node_num; ++q){
394         for (i=0; i<node_num; i++){
395           for (j=0; j<node_num; j++){
396             routingmatrix[p][q][i][j] = x[ptr];
397             ptr++;
398           }
399         }
400       }
401     }
402
403     /* Display the maximum bound violation. */
404     /* status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS); */
405     /* if ( status ) { */
406     /*   fprintf (stderr, "Failed to obtain bound violation.\n"); */
407     /*   goto TERMINATE; */
408     /* } */
409     /* printf ("Maximum bound violation = %17.10g\n", maxviol); */
410
411
412    TERMINATE:
413     free_and_null ((char **) &cstat);
414     free_and_null ((char **) &rstat);
415     free_and_null ((char **) &x);
416
417     free(rhs);
418     free(lb);
```

```
419     free(ub);
420     free(obj);
421     free(sense);
422     free(rowlist);
423     free(collist);
424     free(vallist);
425     for (i=0; i<row_num; ++i){
426       free(rowname[i]);
427     }
428     free(rowname);
429     for (i=0; i<col_num; ++i){
430       free(colname[i]);
431     }
432     free(colname);
433
434     if ( lp != NULL ) {
435       status = CPXfreeprob (env, &lp);
436       if ( status ) {
437         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
438       }
439     }
440
441     /* Free up the CPLEX environment, if necessary */
442
443     if ( env != NULL ) {
444       status = CPXcloseCPLEX (&env);
445
446       /* Note that CPXcloseCPLEX produces no output,
447          so the only way to see the cause of the error is to use
448          CPXgeterrorstring.  For other CPLEX routines, the errors will
449          be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
450
451       if ( status ) {
452         char errmsg[CPXMESSAGEBUFSIZE];
453         fprintf (stderr, "Could not close CPLEX environment.\n");
454         CPXgeterrorstring (env, status, errmsg);
455         fprintf (stderr, "%s", errmsg);
456       }
457     }
458
459     return status;
460
461   }
462
463
464   double
465   opt_sospf_split_hose (const int        node_num,
466                         unsigned int   **nexthopmatrix,
467                         unsigned int  ***ancestormatrix,
468                         unsigned int   **capacitymatrix,
469                         unsigned int    *alpha,
470                         unsigned int    *beta,
471                         double       ****routingmatrix)
472   {
473     unsigned long col_num     =
474       1 +
475       node_num * node_num * node_num * node_num +
476       node_num * node_num * node_num +
477       node_num * node_num * node_num;
478     unsigned long row_num     =
```

```
479       node_num * (node_num-1) * (node_num-2) +     /* internal */
480       node_num * (node_num-1) +                    /* source */
481       (node_num * node_num) +                      /* capacity */
482       (node_num * node_num * node_num * node_num); /* dual */
483     unsigned long nonzero_num =
484       (node_num+1)*(node_num*(node_num-1)*(node_num-2)) +
485       (2*(node_num-1))*(node_num*(node_num-1)) +
486       (node_num*node_num+1)*(node_num*node_num) +
487       (3)*(node_num * node_num * node_num * node_num);
488
489     double *rhs;
490     double *lb;
491     double *ub;
492     double *obj;
493     char   *sense;
494     char   **rowname;
495     char   **colname;
496     int    *rowlist;
497     int    *collist;
498     double *vallist;
499
500     int i,j,p,q;
501     unsigned long ptr = 0;
502     unsigned long row = 0;
503
504     /* double  objval, maxviol; */
505     int     cur_numrows, cur_numcols;
506     double *x     = NULL;
507     int    *cstat = NULL;
508     int    *rstat = NULL;
509
510     CPXENVptr env    = NULL;
511     CPXLPptr  lp     = NULL;
512     int       status = 0;
513     int       solnstat, solnmethod, solntype;
514
515     double congestion = 0.0;
516     /* char *basismsg; */
517
518
519     if ((rhs = (double *)malloc(col_num * sizeof(double))) == NULL){
520       fprintf(stderr, "Memory allocate error\n");
521       ;
522     }
523
524     if ((lb = (double *)malloc(col_num * sizeof(double))) == NULL){
525       fprintf(stderr, "Memory allocate error\n");
526       ;
527     }
528
529     if ((ub = (double *)malloc(col_num * sizeof(double))) == NULL){
530       fprintf(stderr, "Memory allocate error\n");
531       ;
532     }
533
534     if ((obj = (double *)malloc(col_num * sizeof(double))) == NULL){
535       fprintf(stderr, "Memory allocate error\n");
536       ;
537     }
538
```

```
539     if ((sense = (char *)malloc(row_num * sizeof(char))) == NULL){
540       fprintf(stderr, "Memory allocate error\n");
541       ;
542     }
543
544     if ((rowlist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
545       fprintf(stderr, "Memory allocate error\n");
546       ;
547     }
548
549     if ((collist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
550       fprintf(stderr, "Memory allocate error\n");
551       ;
552     }
553
554     if ((vallist = (double *)malloc(nonzero_num * sizeof(double))) == NULL){
555       fprintf(stderr, "Memory allocate error\n");
556       ;
557     }
558
559     if ((rowname = (char **)malloc(row_num * sizeof(char *))) == NULL){
560       fprintf(stderr, "Memory allocate error\n");
561       ;
562     }
563
564     for (i=0; i<row_num; ++i){
565       if ((rowname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
566         fprintf(stderr, "Memory allocate error\n");
567         ;
568       }
569     }
570
571     if ((colname = (char **)malloc(col_num * sizeof(char *))) == NULL){
572       fprintf(stderr, "Memory allocate error\n");
573       ;
574     }
575
576     for (i=0; i<col_num; ++i){
577       if ((colname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
578         fprintf(stderr, "Memory allocate error\n");
579         ;
580       }
581     }
582
583
584     for (i=0; i<node_num; ++i){
585       for (p=0; p<node_num; ++p){
586         for (q=0; q<node_num; ++q){
587           if ((i!=p)&&(i!=q)&&(p!=q)){
588             sprintf(rowname[row], "INTERNAL(%d,%d,%d)", i, p, q);
589             sense[row] = 'E';
590             rhs[row]   = 0.0;
591             row++;
592           }
593         }
594       }
595     }
596
597     for (p=0; p<node_num; ++p){
598       for (q=0; q<node_num; ++q){
```

```
599         if ((p!=q)){
600           sprintf(rowname[row], "SOURCE(%d,%d,%d)", p, p, q);
601           sense[row] = 'E';
602           rhs[row]   = 1.0;
603           row++;
604         }
605       }
606     }
607
608     for (p=0; p<node_num; ++p){
609       for (q=0; q<node_num; ++q){
610         sprintf(rowname[row], "CAPACITY(%d,%d)", p, q);
611         sense[row] = 'L';
612         rhs[row]   = 0.0;
613         row++;
614       }
615     }
616
617     for (p=0; p<node_num; ++p){
618       for (q=0; q<node_num; ++q){
619         for (i=0; i<node_num; ++i){
620           for (j=0; j<node_num; ++j){
621             sprintf(rowname[row], "DUAL(%d,%d,%d,%d)", p, q, i, j);
622             sense[row] = 'L';
623             rhs[row]   = 0.0;
624             row++;
625           }
626         }
627       }
628     }
629
630     ptr      = 0;
631     obj[ptr] = 1.0;
632     ub[ptr]  = CPX_INFBOUND;
633     lb[ptr]  = 0.0;
634     sprintf(colname[ptr], "r");
635     ptr++;
636     for (i=0; i<node_num; ++i){
637       for (j=0; j<node_num; ++j){
638         for (p=0; p<node_num; ++p){
639           for (q=0; q<node_num; ++q){
640             obj[ptr] = 0.0;
641             ub[ptr]  = 1.0;
642             lb[ptr]  = 0.0;
643             sprintf(colname[ptr], "x(%d,%d,%d,%d)", i, j, p, q);
644             ptr++;
645           }
646         }
647       }
648     }
649     for (i=0; i<node_num; ++i){
650       for (j=0; j<node_num; ++j){
651         for (p=0; p<node_num; ++p){
652           obj[ptr] = 0.0;
653           ub[ptr]  = CPX_INFBOUND;
654           lb[ptr]  = 0.0;
655           sprintf(colname[ptr], "lambda(%d,%d,%d)", i, j, p);
656           ptr++;
657         }
658       }
```

```
659     }
660     for (i=0; i<node_num; ++i){
661       for (j=0; j<node_num; ++j){
662         for (p=0; p<node_num; ++p){
663           obj[ptr] = 0.0;
664           ub[ptr]  = CPX_INFBOUND;
665           lb[ptr]  = 0.0;
666           sprintf(colname[ptr], "pai(%d,%d,%d)", i, j, p);
667           ptr++;
668         }
669       }
670     }
671
672
673     ptr = 0;
674     row = 0;
675     for (i=0; i<node_num; ++i){
676       for (p=0; p<node_num; ++p){
677         for (q=0; q<node_num; ++q){
678           if ((i!=p)&&(i!=q)&&(p!=q)){
679             rowlist[ptr] = row;
680             collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
681   i*node_num + nexthopmatrix[q][i] + 1;
682             vallist[ptr] = 1.0;
683             ptr++;
684             for (j = 0; j<node_num; ++j){
685               if (j!=nexthopmatrix[q][i]){
686                 rowlist[ptr] = row;
687                 collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
688   j*node_num + i + 1;
689                 vallist[ptr] = -1.0;
690                 ptr++;
691               }
692             }
693             row++;
694           }
695         }
696       }
697     }
698
699     for (p=0; p<node_num; ++p){
700       for (q=0; q<node_num; ++q){
701         if ((p != q)){
702           for (j=0; j<node_num; j++){
703             if ((ancestormatrix[q][p][j] == 0) && (j != p)){
704               rowlist[ptr] = row;
705               collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
706   p*node_num + j + 1;
707               vallist[ptr] = 1.0;
708               ptr++;
709             }
710           }
711           for (j=0; j<node_num; j++){
712             if (j != p){
713               rowlist[ptr] = row;
714               collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
715   j*node_num + p + 1;
716               vallist[ptr] = -1.0;
717               ptr++;
718             }
```

```
719           }
720           row++;
721         }
722       }
723     }
724
725     for (i=0; i<node_num; ++i){
726       for (j=0; j<node_num; ++j){
727         rowlist[ptr] = row;
728         collist[ptr] = 0;
729         vallist[ptr] = -1.0 * capacitymatrix[i][j];
730         ptr++;
731         for (p=0; p<node_num; ++p){
732           rowlist[ptr] = row;
733           collist[ptr] = i*node_num*node_num + j*node_num + p +
734     (node_num*node_num*node_num*node_num+1);
735           vallist[ptr] = alpha[p];
736           ptr++;
737         }
738         for (p=0; p<node_num; ++p){
739           rowlist[ptr] = row;
740           collist[ptr] = i*node_num*node_num + j*node_num + p +
741     (node_num*node_num*node_num+node_num*node_num*node_num*node_num+1);
742           vallist[ptr] = beta[p];
743           ptr++;
744         }
745         row++;
746       }
747     }
748
749     for (p=0; p<node_num; ++p){
750       for (q=0; q<node_num; ++q){
751         for (i=0; i<node_num; ++i){
752           for (j=0; j<node_num; ++j){
753             /* x */
754             rowlist[ptr] = row;
755             collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
756     i*node_num + j + 1;
757             vallist[ptr] = 1.0;
758             ptr++;
759
760             /* lambda */
761             rowlist[ptr] = row;
762             collist[ptr] = i*node_num*node_num + j*node_num + p +
763     (node_num*node_num*node_num*node_num+1);
764             vallist[ptr] = -1.0;
765             ptr++;
766
767             /* pai */
768             rowlist[ptr] = row;
769             collist[ptr] = i*node_num*node_num + j*node_num + q +
770     (node_num*node_num*node_num+node_num*node_num*node_num*node_num+1);
771             vallist[ptr] = -1.0;
772             ptr++;
773             row++;
774           }
775         }
776       }
777     }
778
```

```
779
780
781     if (ptr > nonzero_num){
782       fprintf (stderr, "non zero number is small!");
783       ;
784     }
785
786     env = CPXopenCPLEX(&status);
787
788     if (env == NULL){
789       char  errmsg[CPXMESSAGEBUFSIZE];
790       fprintf (stderr, "Could not open CPLEX environment.\n");
791       CPXgeterrorstring (env, status, errmsg);
792       fprintf (stderr, "%s", errmsg);
793       ;
794     }
795
796     /* status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON); */
797     /* if ( status ) { */
798     /*   fprintf (stderr, "Failure to turn on screen indicator, error %d.\n", status); */
799     /*     */
800     /* } */
801
802     lp = CPXcreateprob (env, &status, "SOSPFSPLIT");
803     if ( lp == NULL ) {
804       fprintf (stderr, "Failed to create LP.\n");
805       ;
806     }
807
808     CPXchgobjsen (env, lp, CPX_MIN);  /* Problem is minimize */
809
810     status = CPXnewrows (env, lp, row_num, rhs, sense, NULL, rowname);
811     if ( status ) {
812       fprintf (stderr, "Failed to newrows.\n");
813       ;
814     }
815
816     status = CPXnewcols (env, lp, col_num, obj, lb, ub, NULL, colname);
817     if ( status ) {
818       fprintf (stderr, "Failed to newcols.\n");
819       ;
820     }
821
822     status = CPXchgcoeflist (env, lp, ptr, rowlist, collist, vallist);
823     if ( status ) {
824       fprintf (stderr, "Failed to chgcoeflist.\n");
825       ;
826     }
827
828     /* status = CPXwriteprob(env, lp, "check.lp", NULL); */
829     /* if ( status ) { */
830     /*   fprintf (stderr, "Failed to write prob.\n"); */
831     /*     */
832     /* } */
833
834     status   = CPXlpopt(env, lp);
835     solnstat = CPXgetstat (env, lp);
836
837     if      ( solnstat == CPX_STAT_UNBOUNDED ) {
838       printf ("Model is unbounded\n");
```

```
839       ;
840     }
841     else if ( solnstat == CPX_STAT_INFEASIBLE ) {
842       printf ("Model is infeasible\n");
843       ;
844     }
845     else if ( solnstat == CPX_STAT_INForUNBD ) {
846       printf ("Model is infeasible or unbounded\n");
847       ;
848     }
849
850     status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
851     if ( status ) {
852       fprintf (stderr, "Failed to obtain solution info.\n");
853       ;
854     }
855     /* printf ("Solution status %d, solution method %d\n", solnstat, solnmethod); */
856
857     if ( solntype == CPX_NO_SOLN ) {
858       fprintf (stderr, "Solution not available.\n");
859       ;
860     }
861
862     /* status = CPXgetobjval (env, lp, &objval); */
863     /* if ( status ) { */
864     /*   fprintf (stderr, "Failed to obtain objective value.\n"); */
865     /*     */
866     /* } */
867     /* printf ("Objective value %.10g.\n", objval); */
868
869     cur_numcols = CPXgetnumcols (env, lp);
870     cur_numrows = CPXgetnumrows (env, lp);
871
872     /* Retrieve basis, if one is available */
873
874     if ( solntype == CPX_BASIC_SOLN ) {
875       cstat = (int *) malloc (cur_numcols*sizeof(int));
876       rstat = (int *) malloc (cur_numrows*sizeof(int));
877       if ( cstat == NULL || rstat == NULL ) {
878         fprintf (stderr, "No memory for basis statuses.\n");
879         ;
880       }
881
882       status = CPXgetbase (env, lp, cstat, rstat);
883       if ( status ) {
884         fprintf (stderr, "Failed to get basis; error %d.\n", status);
885         ;
886       }
887     }
888     else {
889       printf ("No basis available\n");
890     }
891
892     /* Retrieve solution vector */
893     x = (double *) malloc (cur_numcols*sizeof(double));
894     if ( x == NULL ) {
895       fprintf (stderr, "No memory for solution.\n");
896       ;
897     }
898
```

```
899     status = CPXgetx (env, lp, x, 0, cur_numcols-1);
900     if ( status ) {
901       fprintf (stderr, "Failed to obtain primal solution.\n");
902       ;
903     }
904
905     congestion = x[0];
906     ptr = 1;
907     for (p=0; p<node_num; ++p){
908       for (q=0; q<node_num; ++q){
909         for (i=0; i<node_num; i++){
910           for (j=0; j<node_num; j++){
911             routingmatrix[p][q][i][j] = x[ptr];
912             ptr++;
913           }
914         }
915       }
916     }
917
918     /* Display the maximum bound violation. */
919     /* status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS); */
920     /* if ( status ) { */
921     /*   fprintf (stderr, "Failed to obtain bound violation.\n"); */
922     /*   goto TERMINATE; */
923     /* } */
924     /* printf ("Maximum bound violation = %17.10g\n", maxviol); */
925
926
927    TERMINATE:
928     free_and_null ((char **) &cstat);
929     free_and_null ((char **) &rstat);
930     free_and_null ((char **) &x);
931
932     free(rhs);
933     free(lb);
934     free(ub);
935     free(obj);
936     free(sense);
937     free(rowlist);
938     free(collist);
939     free(vallist);
940     for (i=0; i<row_num; ++i){
941       free(rowname[i]);
942     }
943     free(rowname);
944     for (i=0; i<col_num; ++i){
945       free(colname[i]);
946     }
947     free(colname);
948
949     if ( lp != NULL ) {
950       status = CPXfreeprob (env, &lp);
951       if ( status ) {
952         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
953       }
954     }
955
956     /* Free up the CPLEX environment, if necessary */
957
958     if ( env != NULL ) {
```

```
959       status = CPXcloseCPLEX (&env);
960
961       /* Note that CPXcloseCPLEX produces no output,
962          so the only way to see the cause of the error is to use
963          CPXgeterrorstring.  For other CPLEX routines, the errors will
964          be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
965
966       if ( status ) {
967         char errmsg[CPXMESSAGEBUFSIZE];
968         fprintf (stderr, "Could not close CPLEX environment.\n");
969         CPXgeterrorstring (env, status, errmsg);
970         fprintf (stderr, "%s", errmsg);
971       }
972     }
973
974     return (status < 0 ? (double)status : congestion);
975
976   }
977
978   double
979   opt_traffic_flow_hose (const int        node_num,
980                          unsigned int     link_i,
981                          unsigned int     link_j,
982                          unsigned int    *alpha,
983                          unsigned int    *beta,
984                          double       ****routingmatrix)
985   {
986     unsigned long col_num     =
987       node_num*node_num;
988     unsigned long row_num     =
989       node_num +                  /* alpha */
990       node_num ;                  /* beta */
991     unsigned long nonzero_num =
992       node_num * node_num +
993       node_num * node_num;
994
995     double *rhs;
996     double *lb;
997     double *ub;
998     double *obj;
999     char   *sense;
1000    char   **rowname;
1001    char   **colname;
1002    int    *rowlist;
1003    int    *collist;
1004    double *vallist;
1005
1006    int i,j;
1007    unsigned long ptr = 0;
1008    unsigned long row = 0;
1009
1010    /* double  objval, maxviol; */
1011    int     cur_numrows, cur_numcols;
1012    double *x     = NULL;
1013    int    *cstat = NULL;
1014    int    *rstat = NULL;
1015
1016    CPXENVptr env    = NULL;
1017    CPXLPptr  lp     = NULL;
1018    int       status = 0;
```

```
1019    int       solnstat, solnmethod, solntype;
1020
1021    double traffic = 0.0;
1022
1023    /* char *basismsg; */
1024
1025    if ((rhs = (double *)malloc(col_num * sizeof(double))) == NULL){
1026      fprintf(stderr, "Memory allocate error\n");
1027
1028    }
1029
1030    if ((lb = (double *)malloc(col_num * sizeof(double))) == NULL){
1031      fprintf(stderr, "Memory allocate error\n");
1032
1033    }
1034
1035    if ((ub = (double *)malloc(col_num * sizeof(double))) == NULL){
1036      fprintf(stderr, "Memory allocate error\n");
1037
1038    }
1039
1040    if ((obj = (double *)malloc(col_num * sizeof(double))) == NULL){
1041      fprintf(stderr, "Memory allocate error\n");
1042
1043    }
1044
1045    if ((sense = (char *)malloc(row_num * sizeof(char))) == NULL){
1046      fprintf(stderr, "Memory allocate error\n");
1047
1048    }
1049
1050    if ((rowlist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
1051      fprintf(stderr, "Memory allocate error\n");
1052
1053    }
1054
1055    if ((collist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
1056      fprintf(stderr, "Memory allocate error\n");
1057
1058    }
1059
1060    if ((vallist = (double *)malloc(nonzero_num * sizeof(double))) == NULL){
1061      fprintf(stderr, "Memory allocate error\n");
1062
1063    }
1064
1065    if ((rowname = (char **)malloc(row_num * sizeof(char *))) == NULL){
1066      fprintf(stderr, "Memory allocate error\n");
1067
1068    }
1069
1070    for (i=0; i<row_num; ++i){
1071      if ((rowname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1072        fprintf(stderr, "Memory allocate error\n");
1073
1074      }
1075    }
1076
1077    if ((colname = (char **)malloc(col_num * sizeof(char *))) == NULL){
1078      fprintf(stderr, "Memory allocate error\n");
```

```
1079
1080      }
1081
1082      for (i=0; i<col_num; ++i){
1083        if ((colname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1084          fprintf(stderr, "Memory allocate error\n");
1085
1086        }
1087      }
1088
1089      /* alpha */
1090      for (i=0; i<node_num; ++i){
1091        sprintf (rowname[row], "ALPHA(%d)", i);
1092        sense[row] = 'L';
1093        rhs[row] = alpha[i];
1094        row++;
1095      }
1096
1097      /* beta */
1098      for (i=0; i<node_num; ++i){
1099        sprintf (rowname[row], "BETA(%d)", i);
1100        sense[row] = 'L';
1101        rhs[row] = beta[i];
1102        row++;
1103      }
1104
1105      ptr = 0;
1106      for (i=0; i<node_num; ++i){
1107        for (j=0; j<node_num; ++j){
1108          obj[ptr] = routingmatrix[i][j][link_i][link_j];
1109          ub[ptr]  = CPX_INFBOUND;
1110          lb[ptr]  = 0.0;
1111          sprintf(colname[ptr], "d(%d,%d)", i, j);
1112          ptr++;
1113        }
1114      }
1115
1116      ptr = 0;
1117      row = 0;
1118      /* alpha */
1119      for (i=0; i<node_num; ++i){
1120        for (j=0; j<node_num; ++j){
1121          rowlist[ptr] = row;
1122          collist[ptr] = i*node_num + j;
1123          vallist[ptr] = 1.0;
1124          ptr++;
1125        }
1126        row++;
1127      }
1128
1129      /* beta */
1130      for (j=0; j<node_num; ++j){
1131        for (i=0; i<node_num; ++i){
1132          rowlist[ptr] = row;
1133          collist[ptr] = i*node_num + j;
1134          vallist[ptr] = 1.0;
1135          ptr++;
1136        }
1137        row++;
1138      }
```

```
1139
1140    if (ptr > nonzero_num){
1141      fprintf (stderr, "non zero number is small!");
1142
1143    }
1144
1145    if (row > row_num){
1146      fprintf (stderr, "row number is small!");
1147
1148    }
1149
1150    env = CPXopenCPLEX(&status);
1151
1152    if (env == NULL){
1153      char  errmsg[CPXMESSAGEBUFSIZE];
1154      fprintf (stderr, "Could not open CPLEX environment.\n");
1155      CPXgeterrorstring (env, status, errmsg);
1156      fprintf (stderr, "%s", errmsg);
1157
1158    }
1159
1160    /* status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON); */
1161    /* if ( status ) { */
1162    /*   fprintf (stderr, "Failure to turn on screen indicator, error %d.\n", status); */
1163    /*     */
1164    /* } */
1165
1166    lp = CPXcreateprob (env, &status, "MPLS");
1167    if ( lp == NULL ) {
1168      fprintf (stderr, "Failed to create LP.\n");
1169
1170    }
1171
1172    CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximize */
1173
1174    status = CPXnewrows (env, lp, row_num, rhs, sense, NULL, rowname);
1175    if ( status ) {
1176      fprintf (stderr, "Failed to newrows.\n");
1177
1178    }
1179
1180    status = CPXnewcols (env, lp, col_num, obj, lb, ub, NULL, colname);
1181    if ( status ) {
1182      fprintf (stderr, "Failed to newcols.\n");
1183
1184    }
1185
1186    status = CPXchgcoeflist (env, lp, ptr, rowlist, collist, vallist);
1187    if ( status ) {
1188      fprintf (stderr, "Failed to chgcoeflist.\n");
1189
1190    }
1191
1192    /* status = CPXwriteprob(env, lp, "check_tr.lp", NULL); */
1193    /* if ( status ) { */
1194    /*   fprintf (stderr, "Failed to write prob.\n"); */
1195    /*     */
1196    /* } */
1197
1198    status   = CPXlpopt(env, lp);
```

```
1199    solnstat = CPXgetstat (env, lp);
1200
1201    if      ( solnstat == CPX_STAT_UNBOUNDED ) {
1202      fprintf (stderr,"Model is unbounded\n");
1203
1204    }
1205    else if ( solnstat == CPX_STAT_INFEASIBLE ) {
1206      fprintf (stderr,"Model is infeasible\n");
1207
1208    }
1209    else if ( solnstat == CPX_STAT_INForUNBD ) {
1210      fprintf (stderr,"Model is infeasible or unbounded\n");
1211
1212    }
1213
1214    status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
1215    if ( status ) {
1216      fprintf (stderr, "Failed to obtain solution info.\n");
1217
1218    }
1219    /* fprintf (stderr,"Solution status %d, solution method %d\n", solnstat, solnmethod); */
1220
1221    if ( solntype == CPX_NO_SOLN ) {
1222      fprintf (stderr, "Solution not available.\n");
1223
1224    }
1225
1226    /* status = CPXgetobjval (env, lp, &objval); */
1227    /* if ( status ) { */
1228    /*   fprintf (stderr, "Failed to obtain objective value.\n"); */
1229    /*     */
1230    /* } */
1231    /* fprintf (stderr,"Objective value %.10g.\n", objval); */
1232
1233    cur_numcols = CPXgetnumcols (env, lp);
1234    cur_numrows = CPXgetnumrows (env, lp);
1235
1236    /* Retrieve basis, if one is available */
1237
1238    if ( solntype == CPX_BASIC_SOLN ) {
1239      cstat = (int *) malloc (cur_numcols*sizeof(int));
1240      rstat = (int *) malloc (cur_numrows*sizeof(int));
1241      if ( cstat == NULL || rstat == NULL ) {
1242        fprintf (stderr, "No memory for basis statuses.\n");
1243
1244      }
1245
1246      status = CPXgetbase (env, lp, cstat, rstat);
1247      if ( status ) {
1248        fprintf (stderr, "Failed to get basis; error %d.\n", status);
1249
1250      }
1251    }
1252    else {
1253      fprintf (stderr,"No basis available\n");
1254    }
1255
1256    /* Retrieve solution vector */
1257    x = (double *) malloc (cur_numcols*sizeof(double));
1258    if ( x == NULL ) {
```

```
1259       fprintf (stderr, "No memory for solution.\n");
1260
1261     }
1262
1263     status = CPXgetx (env, lp, x, 0, cur_numcols-1);
1264     if ( status ) {
1265       fprintf (stderr, "Failed to obtain primal solution.\n");
1266
1267     }
1268
1269     for (i=0; i<cur_numcols; ++i){
1270       traffic += x[i];
1271     }
1272
1273     /* Display the maximum bound violation. */
1274     /* status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS); */
1275     /* if ( status ) { */
1276     /*   fprintf (stderr, "Failed to obtain bound violation.\n"); */
1277     /*     */
1278     /* } */
1279     /* printf ("Maximum bound violation = %17.10g\n", maxviol); */
1280
1281
1282    TERMINATE:
1283     free_and_null ((char **) &cstat);
1284     free_and_null ((char **) &rstat);
1285     free_and_null ((char **) &x);
1286
1287     free(rhs);
1288     free(lb);
1289     free(ub);
1290     free(obj);
1291     free(sense);
1292     free(rowlist);
1293     free(collist);
1294     free(vallist);
1295     for (i=0; i<row_num; ++i){
1296       free(rowname[i]);
1297     }
1298     free(rowname);
1299     for (i=0; i<col_num; ++i){
1300       free(colname[i]);
1301     }
1302     free(colname);
1303
1304     if ( lp != NULL ) {
1305       status = CPXfreeprob (env, &lp);
1306       if ( status ) {
1307         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
1308       }
1309     }
1310
1311     /* Free up the CPLEX environment, if necessary */
1312
1313     if ( env != NULL ) {
1314       status = CPXcloseCPLEX (&env);
1315
1316       /* Note that CPXcloseCPLEX produces no output,
1317          so the only way to see the cause of the error is to use
1318          CPXgeterrorstring.  For other CPLEX routines, the errors will
```

```
1319          be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
1320
1321       if ( status ) {
1322         char errmsg[CPXMESSAGEBUFSIZE];
1323         fprintf (stderr, "Could not close CPLEX environment.\n");
1324         CPXgeterrorstring (env, status, errmsg);
1325         fprintf (stderr, "%s", errmsg);
1326       }
1327     }
1328
1329     return (status < 0 ? (double)status : traffic);
1330
1331   }
1332
1333   double
1334   opt_sospf_local_reroute_pipe (const int      node_num,
1335                                 int            b_num,
1336                                 unsigned int   b_i,
1337                                 unsigned int   b_j,
1338                                 unsigned int   **nexthopmatrix,
1339                                 unsigned int  ***ancestormatrix,
1340                                 unsigned int   **capacitymatrix,
1341                                 unsigned int   **trafficmatrix,
1342                                 double       ****routingmatrix)
1343   {
1344     unsigned long col_num = node_num * node_num * node_num * node_num + 1;
1345     unsigned long row_num =
1346       node_num * (node_num-1) * (node_num-2) +
1347       node_num * (node_num-1) +
1348       (node_num * node_num) +
1349       b_num;
1350     unsigned long nonzero_num =
1351       (node_num+1)*(node_num*(node_num-1)*(node_num-2)) +
1352       (2*(node_num-1))*(node_num*(node_num-1)) +
1353       (node_num*node_num+1)*(node_num*node_num) +
1354       node_num * node_num * node_num * node_num - node_num * node_num;
1355
1356     double *rhs; //right hand side
1357     double *lb; //lower bound
1358     double *ub; //upper bound
1359     double *obj;
1360     char   *sense;
1361     char   **rowname;
1362     char   **colname;
1363     int    *rowlist;
1364     int    *collist;
1365     double *vallist;
1366     double congestion;
1367
1368     int i,j,p,q;
1369     unsigned long ptr = 0;
1370     unsigned long row = 0;
1371
1372     /* double  objval, maxviol; */
1373     int     cur_numrows, cur_numcols;
1374     double *x     = NULL;
1375     int    *cstat = NULL;
1376     int    *rstat = NULL;
1377
1378     CPXENVptr env    = NULL;
```

```
1379    CPXLPptr  lp     = NULL;
1380    int       status = 0;
1381    int       solnstat, solnmethod, solntype;
1382
1383    /* char *basismsg; */
1384
1385
1386    if ((rhs = (double *)malloc(row_num * sizeof(double))) == NULL){
1387      status = -1;
1388      fprintf(stderr, "Memory allocate error\n");
1389      goto TERMINATE;
1390    }
1391
1392    if ((lb = (double *)malloc(col_num * sizeof(double))) == NULL){
1393      status = -1;
1394      fprintf(stderr, "Memory allocate error\n");
1395      goto TERMINATE;
1396    }
1397
1398    if ((ub = (double *)malloc(col_num * sizeof(double))) == NULL){
1399      status = -1;
1400      fprintf(stderr, "Memory allocate error\n");
1401      goto TERMINATE;
1402    }
1403
1404    if ((obj = (double *)malloc(col_num * sizeof(double))) == NULL){
1405      status = -1;
1406      fprintf(stderr, "Memory allocate error\n");
1407      goto TERMINATE;
1408    }
1409
1410    if ((sense = (char *)malloc(row_num * sizeof(char))) == NULL){
1411      status = -1;
1412      fprintf(stderr, "Memory allocate error\n");
1413      goto TERMINATE;
1414    }
1415
1416    if ((rowlist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
1417      status = -1;
1418      fprintf(stderr, "Memory allocate error\n");
1419      goto TERMINATE;
1420    }
1421
1422    if ((collist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
1423      status = -1;
1424      fprintf(stderr, "Memory allocate error\n");
1425      goto TERMINATE;
1426    }
1427
1428    if ((vallist = (double *)malloc(nonzero_num * sizeof(double))) == NULL){
1429      status = -1;
1430      fprintf(stderr, "Memory allocate error\n");
1431      goto TERMINATE;
1432    }
1433
1434    if ((rowname = (char **)malloc(row_num * sizeof(char *))) == NULL){
1435      status = -1;
1436      fprintf(stderr, "Memory allocate error\n");
1437      goto TERMINATE;
1438    }
```

```
1439    for (i=0; i<row_num; ++i){
1440      if ((rowname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1441        status = -1;
1442        fprintf(stderr, "Memory allocate error\n");
1443        goto TERMINATE;
1444      }
1445    }
1446
1447    if ((colname = (char **)malloc(col_num * sizeof(char *))) == NULL){
1448      status = -1;
1449      fprintf(stderr, "Memory allocate error\n");
1450      goto TERMINATE;
1451    }
1452    for (i=0; i<col_num; ++i){
1453      if ((colname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1454        status = -1;
1455        fprintf(stderr, "Memory allocate error\n");
1456        goto TERMINATE;
1457      }
1458    }
1459
1460
1461    for (i=0; i<node_num; ++i){
1462      for (p=0; p<node_num; ++p){
1463        for (q=0; q<node_num; ++q){
1464          if ((i!=p)&&(i!=q)&&(p!=q)){
1465            sprintf(rowname[row], "INTERNAL(%d,%d,%d)", i, p, q);
1466            sense[row] = 'E';
1467            rhs[row]   = 0.0;
1468            row++;
1469          }
1470        }
1471      }
1472    }
1473
1474    for (p=0; p<node_num; ++p){
1475      for (q=0; q<node_num; ++q){
1476        if ((p!=q)){
1477          sprintf(rowname[row], "SOURCE(%d,%d,%d)", p, p, q);
1478          sense[row] = 'E';
1479          rhs[row]   = 1.0;
1480          row++;
1481        }
1482      }
1483    }
1484
1485    for (i=0; i<node_num; ++i){
1486      for (j=0; j<node_num; ++j){
1487        sprintf(rowname[row], "CAPACITY(%d,%d)", i, j);
1488        sense[row] = 'L';
1489        rhs[row]   = 0.0;
1490        row++;
1491      }
1492    }
1493
1494    for (p=0; p<node_num; ++p){
1495      for (q=0; q<node_num; ++q){
1496        if((routingmatrix[p][q][b_i][b_j]==0)&&(routingmatrix[p][q][b_j][b_i]==0)){
1497          for (i=0; i<node_num; ++i){
1498            if(i==p){
```

```
1499              for (j=0; j<node_num; ++j){
1500                sprintf(rowname[row], "FIX(%d,%d,%d,%d)",p, q, i, j);
1501                sense[row] = 'E';
1502                rhs[row]   = (double)routingmatrix[p][q][i][j];
1503                row++;
1504              }
1505            }
1506          }
1507        }
1508      }
1509    }
1510
1511    ptr      = 0;
1512    obj[ptr] = 1.0;
1513    ub[ptr]  = CPX_INFBOUND;
1514    lb[ptr]  = 0.0;
1515    sprintf(colname[ptr], "r");
1516    ptr++;
1517    for (p=0; p<node_num; ++p){
1518      for (q=0; q<node_num; ++q){
1519        for (i=0; i<node_num; ++i){
1520          for (j=0; j<node_num; ++j){
1521            obj[ptr] = 0.0;
1522            ub[ptr]  = 1.0;
1523            lb[ptr]  = 0.0;
1524            sprintf(colname[ptr], "x(%d,%d,%d,%d)", p, q, i, j);
1525            ptr++;
1526          }
1527        }
1528      }
1529    }
1530
1531    ptr = 0;
1532    row = 0;
1533    for (p=0; p<node_num; ++p){
1534      for (q=0; q<node_num; ++q){
1535        for (i=0; i<node_num; ++i){
1536          if ((i!=p)&&(i!=q)&&(p!=q)){
1537            rowlist[ptr] = row;
1538            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1539    i*node_num + nexthopmatrix[q][i] + 1;
1540            vallist[ptr] = 1.0;
1541            ptr++;
1542            for (j=0; j<node_num; ++j){
1543              rowlist[ptr] = row;
1544              collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1545    j*node_num + i + 1;
1546              vallist[ptr] = -1.0;
1547              ptr++;
1548            }
1549            row++;
1550          }
1551        }
1552      }
1553    }
1554
1555
1556    for (p=0; p<node_num; ++p){
1557      for (q=0; q<node_num; ++q){
1558        if ((p != q)){
```

```
1559          for (j=0; j<node_num; j++){
1560            if ((ancestormatrix[q][p][j] == 0) && (j != p)){
1561              rowlist[ptr] = row;
1562              collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1563    p*node_num + j + 1;
1564              vallist[ptr] = 1.0;
1565              ptr++;
1566            }
1567          }
1568          for (j=0; j<node_num; j++){
1569            if (j != p){
1570              rowlist[ptr] = row;
1571              collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1572    j*node_num + p + 1;
1573              vallist[ptr] = -1.0;
1574              ptr++;
1575            }
1576          }
1577          row++;
1578        }
1579      }
1580     }
1581
1582
1583     for (i=0; i<node_num; ++i){
1584       for (j=0; j<node_num; ++j){
1585         rowlist[ptr] = row;
1586         collist[ptr] = 0;
1587         vallist[ptr] = -1.0 * capacitymatrix[i][j];
1588         ptr++;
1589         for (p=0; p<node_num; ++p){
1590           for (q=0; q<node_num; ++q){
1591             rowlist[ptr] = row;
1592             collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1593    i*node_num + j + 1;
1594             vallist[ptr] = trafficmatrix[p][q];
1595             ptr++;
1596           }
1597         }
1598         row++;
1599       }
1600     }
1601
1602
1603     for (p=0; p<node_num; ++p){
1604       for (q=0; q<node_num; ++q){
1605         if((routingmatrix[p][q][b_i][b_j]==0)&&(routingmatrix[p][q][b_j][b_i]==0)){
1606           for (i=0; i<node_num; ++i){
1607             if(i==p){
1608               for (j=0; j<node_num; ++j){
1609                 rowlist[ptr] = row;
1610                 collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
1611    i*node_num + j + 1;
1612                 vallist[ptr] = 1.0;
1613                 ptr++;
1614                 row++;
1615               }
1616             }
1617           }
1618         }
```

```
1619        }
1620      }
1621
1622
1623      if (ptr > nonzero_num){
1624        status = -1;
1625        fprintf (stderr, "non zero number is small!");
1626        goto TERMINATE;
1627      }
1628
1629      env = CPXopenCPLEX(&status);
1630
1631      if (env == NULL){
1632        char  errmsg[CPXMESSAGEBUFSIZE];
1633        fprintf (stderr, "Could not open CPLEX environment.\n");
1634        CPXgeterrorstring (env, status, errmsg);
1635        fprintf (stderr, "%s", errmsg);
1636        goto TERMINATE;
1637      }
1638
1639      /* status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON); */
1640      /* if ( status ) { */
1641      /*   fprintf (stderr, "Failure to turn on screen indicator, error %d.\n", status); */
1642      /*   goto TERMINATE; */
1643      /* } */
1644
1645      lp = CPXcreateprob (env, &status, "SOSPFSPLIT");
1646      if ( lp == NULL ) {
1647        fprintf (stderr, "Failed to create LP.\n");
1648        goto TERMINATE;
1649      }
1650
1651      CPXchgobjsen (env, lp, CPX_MIN);  /* Problem is minimize */
1652
1653      status = CPXnewrows (env, lp, row_num, rhs, sense, NULL, rowname);
1654      if ( status ) {
1655        fprintf (stderr, "Failed to newrows.\n");
1656        goto TERMINATE;
1657      }
1658
1659      status = CPXnewcols (env, lp, col_num, obj, lb, ub, NULL, colname);
1660      if ( status ) {
1661        fprintf (stderr, "Failed to newcols.\n");
1662        goto TERMINATE;
1663      }
1664
1665
1666      status = CPXchgcoeflist (env, lp, ptr, rowlist, collist, vallist);
1667      if ( status ) {
1668        fprintf (stderr, "Failed to chgcoeflist.\n");
1669        goto TERMINATE;
1670      }
1671
1672      /* status = CPXwriteprob(env, lp, "check.lp", NULL); */
1673      /* if ( status ) { */
1674      /*   fprintf (stderr, "Failed to write prob.\n"); */
1675      /*   goto TERMINATE; */
1676      /* } */
1677
1678      status   = CPXlpopt(env, lp);
```

```
1679    solnstat = CPXgetstat (env, lp);
1680
1681    if      ( solnstat == CPX_STAT_UNBOUNDED ) {
1682      printf ("Model is unbounded\n");
1683      goto TERMINATE;
1684    }
1685    else if ( solnstat == CPX_STAT_INFEASIBLE ) {
1686      printf ("Model is infeasible\n");
1687      goto TERMINATE;
1688    }
1689    else if ( solnstat == CPX_STAT_INForUNBD ) {
1690      printf ("Model is infeasible or unbounded\n");
1691      goto TERMINATE;
1692    }
1693
1694    status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
1695    if ( status ) {
1696      fprintf (stderr, "Failed to obtain solution info.\n");
1697      goto TERMINATE;
1698    }
1699    /* printf ("Solution status %d, solution method %d\n", solnstat, solnmethod); */
1700
1701    if ( solntype == CPX_NO_SOLN ) {
1702      fprintf (stderr, "Solution not available.\n");
1703      goto TERMINATE;
1704    }
1705
1706    /* status = CPXgetobjval (env, lp, &objval); */
1707    /* if ( status ) { */
1708    /*   fprintf (stderr, "Failed to obtain objective value.\n"); */
1709    /*   goto TERMINATE; */
1710    /* } */
1711    /* printf ("Objective value %.10g.\n", objval); */
1712
1713    cur_numcols = CPXgetnumcols (env, lp);
1714    cur_numrows = CPXgetnumrows (env, lp);
1715
1716    /* Retrieve basis, if one is available */
1717
1718    if ( solntype == CPX_BASIC_SOLN ) {
1719      cstat = (int *) malloc (cur_numcols*sizeof(int));
1720      rstat = (int *) malloc (cur_numrows*sizeof(int));
1721      if ( cstat == NULL || rstat == NULL ) {
1722        fprintf (stderr, "No memory for basis statuses.\n");
1723        goto TERMINATE;
1724      }
1725
1726      status = CPXgetbase (env, lp, cstat, rstat);
1727      if ( status ) {
1728        fprintf (stderr, "Failed to get basis; error %d.\n", status);
1729        goto TERMINATE;
1730      }
1731    }
1732    else {
1733      printf ("No basis available\n");
1734    }
1735
1736    /* Retrieve solution vector */
1737    x = (double *) malloc (cur_numcols*sizeof(double));
1738    if ( x == NULL ) {
```

```
1739      fprintf (stderr, "No memory for solution.\n");
1740      goto TERMINATE;
1741    }
1742
1743    status = CPXgetx (env, lp, x, 0, cur_numcols-1);
1744    if ( status ) {
1745      fprintf (stderr, "Failed to obtain primal solution.\n");
1746      goto TERMINATE;
1747    }
1748
1749    congestion = x[0];
1750    ptr = 1;
1751    /* printf ("b_i=%d, b_j=%d\n", b_i, b_j); */
1752    for (p=0; p<node_num; ++p){
1753      for (q=0; q<node_num; ++q){
1754        for (i=0; i<node_num; i++){
1755          for (j=0; j<node_num; j++){
1756            routingmatrix[p][q][i][j] = x[ptr];
1757            ptr++;
1758          }
1759        }
1760      }
1761    }
1762
1763    /* Display the maximum bound violation. */
1764    /* status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS); */
1765    /* if ( status ) { */
1766    /*   fprintf (stderr, "Failed to obtain bound violation.\n"); */
1767    /*   goto TERMINATE; */
1768    /* } */
1769    /* printf ("Maximum bound violation = %17.10g\n", maxviol); */
1770
1771
1772   TERMINATE:
1773    free_and_null ((char **) &cstat);
1774    free_and_null ((char **) &rstat);
1775    free_and_null ((char **) &x);
1776
1777    free(rhs);
1778    free(lb);
1779    free(ub);
1780    free(obj);
1781    free(sense);
1782    free(rowlist);
1783    free(collist);
1784    free(vallist);
1785    for (i=0; i<row_num; ++i){
1786      free(rowname[i]);
1787    }
1788    free(rowname);
1789    for (i=0; i<col_num; ++i){
1790      free(colname[i]);
1791    }
1792    free(colname);
1793
1794    if ( lp != NULL ) {
1795      status = CPXfreeprob (env, &lp);
1796      if ( status ) {
1797        fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
1798      }
```

```
1799    }
1800
1801    /* Free up the CPLEX environment, if necessary */
1802
1803    if ( env != NULL ) {
1804      status = CPXcloseCPLEX (&env);
1805
1806      /* Note that CPXcloseCPLEX produces no output,
1807         so the only way to see the cause of the error is to use
1808         CPXgeterrorstring.  For other CPLEX routines, the errors will
1809         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
1810
1811      if ( status ) {
1812        char errmsg[CPXMESSAGEBUFSIZE];
1813        fprintf (stderr, "Could not close CPLEX environment.\n");
1814        CPXgeterrorstring (env, status, errmsg);
1815        fprintf (stderr, "%s", errmsg);
1816      }
1817    }
1818
1819    return status;
1820  }
1821
1822
1823  double
1824  opt_sospf_local_reroute_hose (const int      node_num,
1825                                int            b_num,
1826                                unsigned int   b_i,
1827                                unsigned int   b_j,
1828                                unsigned int   **nexthopmatrix,
1829                                unsigned int  ***ancestormatrix,
1830                                unsigned int   **capacitymatrix,
1831                                unsigned int   **trafficmatrix,
1832                                double       ****routingmatrix)
1833  {
1834    unsigned long col_num =
1835      1 +
1836      node_num * node_num * node_num * node_num +
1837      node_num * node_num * node_num +
1838      node_num * node_num * node_num;
1839    unsigned long row_num =
1840      node_num * (node_num-1) * (node_num-2) +      /* internal */
1841      node_num * (node_num-1) +                     /* source */
1842      (node_num * node_num) +                       /* capacity */
1843      (node_num * node_num * node_num * node_num) + /* dual */
1844      b_num;                                        /* stag */
1845    unsigned long nonzero_num =
1846      /* (node_num+1)*(node_num*(node_num-1)*(node_num-2)) + */
1847      /* (2*(node_num-1))*(node_num*(node_num-1)) + */
1848      /* (node_num*node_num+1)*(node_num*node_num) + */
1849      /* node_num * node_num * node_num * node_num - node_num * node_num; */
1850      (node_num+1)*(node_num*(node_num-1)*(node_num-2)) +
1851      (2*(node_num-1))*(node_num*(node_num-1)) +
1852      (node_num*node_num+1)*(node_num*node_num) +
1853      (3)*(node_num * node_num * node_num * node_num);
1854
1855    double *rhs; //right hand side
1856    double *lb; //lower bound
1857    double *ub; //upper bound
1858    double *obj;
```

```
1859     char   *sense;
1860     char   **rowname;
1861     char   **colname;
1862     int    *rowlist;
1863     int    *collist;
1864     double *vallist;
1865     double congestion;
1866
1867     unsigned int    *alpha;
1868     unsigned int    *beta;
1869
1870     int i,j,p,q;
1871     unsigned long ptr = 0;
1872     unsigned long row = 0;
1873
1874     /* double  objval, maxviol; */
1875     int     cur_numrows, cur_numcols;
1876     double *x     = NULL;
1877     int    *cstat = NULL;
1878     int    *rstat = NULL;
1879
1880     CPXENVptr env    = NULL;
1881     CPXLPptr  lp     = NULL;
1882     int       status = 0;
1883     int       solnstat, solnmethod, solntype;
1884
1885     /* char *basismsg; */
1886
1887
1888     if ((rhs = (double *)malloc(row_num * sizeof(double))) == NULL){
1889       status = -1;
1890       fprintf(stderr, "Memory allocate error\n");
1891       goto TERMINATE;
1892     }
1893
1894     if ((lb = (double *)malloc(col_num * sizeof(double))) == NULL){
1895       status = -1;
1896       fprintf(stderr, "Memory allocate error\n");
1897       goto TERMINATE;
1898     }
1899
1900     if ((ub = (double *)malloc(col_num * sizeof(double))) == NULL){
1901       status = -1;
1902       fprintf(stderr, "Memory allocate error\n");
1903       goto TERMINATE;
1904     }
1905
1906     if ((obj = (double *)malloc(col_num * sizeof(double))) == NULL){
1907       status = -1;
1908       fprintf(stderr, "Memory allocate error\n");
1909       goto TERMINATE;
1910     }
1911
1912     if ((sense = (char *)malloc(row_num * sizeof(char))) == NULL){
1913       status = -1;
1914       fprintf(stderr, "Memory allocate error\n");
1915       goto TERMINATE;
1916     }
1917
1918     if ((rowlist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
```

```
1919      status = -1;
1920      fprintf(stderr, "Memory allocate error\n");
1921      goto TERMINATE;
1922    }
1923
1924    if ((collist = (int *)malloc(nonzero_num * sizeof(int))) == NULL){
1925      status = -1;
1926      fprintf(stderr, "Memory allocate error\n");
1927      goto TERMINATE;
1928    }
1929
1930    if ((vallist = (double *)malloc(nonzero_num * sizeof(double))) == NULL){
1931      status = -1;
1932      fprintf(stderr, "Memory allocate error\n");
1933      goto TERMINATE;
1934    }
1935
1936    if ((rowname = (char **)malloc(row_num * sizeof(char *))) == NULL){
1937      status = -1;
1938      fprintf(stderr, "Memory allocate error\n");
1939      goto TERMINATE;
1940    }
1941    for (i=0; i<row_num; ++i){
1942      if ((rowname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1943        status = -1;
1944        fprintf(stderr, "Memory allocate error\n");
1945        goto TERMINATE;
1946      }
1947    }
1948
1949    if ((colname = (char **)malloc(col_num * sizeof(char *))) == NULL){
1950      status = -1;
1951      fprintf(stderr, "Memory allocate error\n");
1952      goto TERMINATE;
1953    }
1954    for (i=0; i<col_num; ++i){
1955      if ((colname[i] = (char *)malloc(30 * sizeof(char *))) == NULL){
1956        status = -1;
1957        fprintf(stderr, "Memory allocate error\n");
1958        goto TERMINATE;
1959      }
1960    }
1961    if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))      == NULL);
1962    memset(alpha, 0, node_num);
1963    if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))       == NULL);
1964    memset(beta, 0, node_num);
1965
1966    /* make alpha & beta */
1967    for (i=0; i<node_num; ++i){
1968      alpha[i] = beta[i] = 0;
1969    }
1970    for (i=0; i<node_num; ++i){
1971      for (j=0; j<node_num; ++j){
1972        alpha[i] += trafficmatrix[i][j];
1973        beta[j] += trafficmatrix[i][j];
1974      }
1975    }
1976
1977    for (i=0; i<node_num; ++i){
1978      for (p=0; p<node_num; ++p){
```

```
1979        for (q=0; q<node_num; ++q){
1980          if ((i!=p)&&(i!=q)&&(p!=q)){
1981            sprintf(rowname[row], "INTERNAL(%d,%d,%d)", i, p, q);
1982            sense[row] = 'E';
1983            rhs[row]   = 0.0;
1984            row++;
1985          }
1986        }
1987      }
1988    }
1989
1990    for (p=0; p<node_num; ++p){
1991      for (q=0; q<node_num; ++q){
1992        if ((p!=q)){
1993          sprintf(rowname[row], "SOURCE(%d,%d,%d)", p, p, q);
1994          sense[row] = 'E';
1995          rhs[row]   = 1.0;
1996          row++;
1997        }
1998      }
1999    }
2000
2001    for (i=0; i<node_num; ++i){
2002      for (j=0; j<node_num; ++j){
2003        sprintf(rowname[row], "CAPACITY(%d,%d)", i, j);
2004        sense[row] = 'L';
2005        rhs[row]   = 0.0;
2006        row++;
2007      }
2008    }
2009
2010    for (p=0; p<node_num; ++p){
2011      for (q=0; q<node_num; ++q){
2012        for (i=0; i<node_num; ++i){
2013          for (j=0; j<node_num; ++j){
2014            sprintf(rowname[row], "DUAL(%d,%d,%d,%d)", p, q, i, j);
2015            sense[row] = 'L';
2016            rhs[row]   = 0.0;
2017            row++;
2018          }
2019        }
2020      }
2021    }
2022
2023    for (p=0; p<node_num; ++p){
2024      for (q=0; q<node_num; ++q){
2025        if((routingmatrix[p][q][b_i][b_j]==0)&&(routingmatrix[p][q][b_j][b_i]==0)){
2026          for (i=0; i<node_num; ++i){
2027            if(i==p){
2028              for (j=0; j<node_num; ++j){
2029                sprintf(rowname[row], "FIX(%d,%d,%d,%d)",p, q, i, j);
2030                sense[row] = 'E';
2031                rhs[row]   = (double)routingmatrix[p][q][i][j];
2032                row++;
2033              }
2034            }
2035          }
2036        }
2037      }
2038    }
```

```
2039
2040     ptr      = 0;
2041     obj[ptr] = 1.0;
2042     ub[ptr]  = CPX_INFBOUND;
2043     lb[ptr]  = 0.0;
2044     sprintf(colname[ptr], "r");
2045     ptr++;
2046     for (p=0; p<node_num; ++p){
2047       for (q=0; q<node_num; ++q){
2048         for (i=0; i<node_num; ++i){
2049           for (j=0; j<node_num; ++j){
2050             obj[ptr] = 0.0;
2051             ub[ptr]  = 1.0;
2052             lb[ptr]  = 0.0;
2053             sprintf(colname[ptr], "x(%d,%d,%d,%d)", p, q, i, j);
2054             ptr++;
2055           }
2056         }
2057       }
2058     }
2059     for (i=0; i<node_num; ++i){
2060       for (j=0; j<node_num; ++j){
2061         for (p=0; p<node_num; ++p){
2062           obj[ptr] = 0.0;
2063           ub[ptr]  = CPX_INFBOUND;
2064           lb[ptr]  = 0.0;
2065           sprintf(colname[ptr], "lambda(%d,%d,%d)", i, j, p);
2066           ptr++;
2067         }
2068       }
2069     }
2070     for (i=0; i<node_num; ++i){
2071       for (j=0; j<node_num; ++j){
2072         for (p=0; p<node_num; ++p){
2073           obj[ptr] = 0.0;
2074           ub[ptr]  = CPX_INFBOUND;
2075           lb[ptr]  = 0.0;
2076           sprintf(colname[ptr], "pai(%d,%d,%d)", i, j, p);
2077           ptr++;
2078         }
2079       }
2080     }
2081
2082
2083     ptr = 0;
2084     row = 0;
2085
2086     for (p=0; p<node_num; ++p){
2087       for (q=0; q<node_num; ++q){
2088         for (i=0; i<node_num; ++i){
2089           if ((i!=p)&&(i!=q)&&(p!=q)){
2090             rowlist[ptr] = row;
2091             collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
2092   i*node_num + nexthopmatrix[q][i] + 1;
2093             vallist[ptr] = 1.0;
2094             ptr++;
2095             for (j=0; j<node_num; ++j){
2096               rowlist[ptr] = row;
2097               collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
2098   j*node_num + i + 1;
```

```
            vallist[ptr] = -1.0;
            ptr++;
          }
          row++;
        }
      }
    }
  }

  for (p=0; p<node_num; ++p){
    for (q=0; q<node_num; ++q){
      if ((p != q)){
        for (j=0; j<node_num; j++){
          if ((ancestormatrix[q][p][j] == 0) && (j != p)){
            rowlist[ptr] = row;
            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
p*node_num + j + 1;
            vallist[ptr] = 1.0;
            ptr++;
          }
        }
        for (j=0; j<node_num; j++){
          if (j != p){
            rowlist[ptr] = row;
            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
j*node_num + p + 1;
            vallist[ptr] = -1.0;
            ptr++;
          }
        }
        row++;
      }
    }
  }

  for (i=0; i<node_num; ++i){
    for (j=0; j<node_num; ++j){
      rowlist[ptr] = row;
      collist[ptr] = 0;
      vallist[ptr] = -1.0 * capacitymatrix[i][j];
      ptr++;
      for (p=0; p<node_num; ++p){
        rowlist[ptr] = row;
        collist[ptr] = i*node_num*node_num + j*node_num + p +
(node_num*node_num*node_num*node_num+1);
        vallist[ptr] = alpha[p];
        ptr++;
      }
      for (p=0; p<node_num; ++p){
        rowlist[ptr] = row;
        collist[ptr] = i*node_num*node_num + j*node_num + p +
(node_num*node_num*node_num+node_num*node_num*node_num*node_num+1);
        vallist[ptr] = beta[p];
        ptr++;
      }
      row++;
    }
  }

  for (p=0; p<node_num; ++p){
```

```
2159      for (q=0; q<node_num; ++q){
2160        for (i=0; i<node_num; ++i){
2161          for (j=0; j<node_num; ++j){
2162            /* x */
2163            rowlist[ptr] = row;
2164            collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
2165    i*node_num + j + 1;
2166            vallist[ptr] = 1.0;
2167            ptr++;
2168
2169            /* lambda */
2170            rowlist[ptr] = row;
2171            collist[ptr] = i*node_num*node_num + j*node_num + p +
2172    (node_num*node_num*node_num*node_num+1);
2173            vallist[ptr] = -1.0;
2174            ptr++;
2175
2176            /* pai */
2177            rowlist[ptr] = row;
2178            collist[ptr] = i*node_num*node_num + j*node_num + q +
2179    (node_num*node_num*node_num+node_num*node_num*node_num*node_num+1);
2180            vallist[ptr] = -1.0;
2181            ptr++;
2182
2183            row++;
2184          }
2185        }
2186      }
2187     }
2188
2189     for (p=0; p<node_num; ++p){
2190       for (q=0; q<node_num; ++q){
2191         if((routingmatrix[p][q][b_i][b_j]==0)&&(routingmatrix[p][q][b_j][b_i]==0)){
2192           for (i=0; i<node_num; ++i){
2193             if(i==p){
2194               for (j=0; j<node_num; ++j){
2195                 rowlist[ptr] = row;
2196                 collist[ptr] = p*node_num*node_num*node_num + q*node_num*node_num +
2197    i*node_num + j + 1;
2198                 vallist[ptr] = 1.0;
2199                 ptr++;
2200                 row++;
2201               }
2202             }
2203           }
2204         }
2205       }
2206     }
2207
2208     if (ptr > nonzero_num){
2209       status = -1;
2210       fprintf (stderr, "non zero number is small!");
2211       goto TERMINATE;
2212     }
2213
2214     env = CPXopenCPLEX(&status);
2215
2216     if (env == NULL){
2217       char  errmsg[CPXMESSAGEBUFSIZE];
2218       fprintf (stderr, "Could not open CPLEX environment.\n");
```

```
2219      CPXgeterrorstring (env, status, errmsg);
2220      fprintf (stderr, "%s", errmsg);
2221      goto TERMINATE;
2222    }
2223
2224    /* status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON); */
2225    /* if ( status ) { */
2226    /*   fprintf (stderr, "Failure to turn on screen indicator, error %d.\n", status); */
2227    /*   goto TERMINATE; */
2228    /* } */
2229
2230    lp = CPXcreateprob (env, &status, "SOSPFSPLIT");
2231    if ( lp == NULL ) {
2232      fprintf (stderr, "Failed to create LP.\n");
2233      goto TERMINATE;
2234    }
2235
2236    CPXchgobjsen (env, lp, CPX_MIN);  /* Problem is minimize */
2237
2238    status = CPXnewrows (env, lp, row_num, rhs, sense, NULL, rowname);
2239    if ( status ) {
2240      fprintf (stderr, "Failed to newrows.\n");
2241      goto TERMINATE;
2242    }
2243
2244    status = CPXnewcols (env, lp, col_num, obj, lb, ub, NULL, colname);
2245    if ( status ) {
2246      fprintf (stderr, "Failed to newcols.\n");
2247      goto TERMINATE;
2248    }
2249
2250
2251    status = CPXchgcoeflist (env, lp, ptr, rowlist, collist, vallist);
2252    if ( status ) {
2253      fprintf (stderr, "Failed to chgcoeflist.\n");
2254      goto TERMINATE;
2255    }
2256
2257    /* status = CPXwriteprob(env, lp, "check.lp", NULL); */
2258    /* if ( status ) { */
2259    /*   fprintf (stderr, "Failed to write prob.\n"); */
2260    /*   goto TERMINATE; */
2261    /* } */
2262
2263    status   = CPXlpopt(env, lp);
2264    solnstat = CPXgetstat (env, lp);
2265
2266    if      ( solnstat == CPX_STAT_UNBOUNDED ) {
2267      printf ("Model is unbounded\n");
2268      goto TERMINATE;
2269    }
2270    else if ( solnstat == CPX_STAT_INFEASIBLE ) {
2271      printf ("Model is infeasible\n");
2272      goto TERMINATE;
2273    }
2274    else if ( solnstat == CPX_STAT_INForUNBD ) {
2275      printf ("Model is infeasible or unbounded\n");
2276      goto TERMINATE;
2277    }
2278
```

```
2279    status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
2280    if ( status ) {
2281      fprintf (stderr, "Failed to obtain solution info.\n");
2282      goto TERMINATE;
2283    }
2284    /* printf ("Solution status %d, solution method %d\n", solnstat, solnmethod); */
2285
2286    if ( solntype == CPX_NO_SOLN ) {
2287      fprintf (stderr, "Solution not available.\n");
2288      goto TERMINATE;
2289    }
2290
2291    /* status = CPXgetobjval (env, lp, &objval); */
2292    /* if ( status ) { */
2293    /*   fprintf (stderr, "Failed to obtain objective value.\n"); */
2294    /*   goto TERMINATE; */
2295    /* } */
2296    /* printf ("Objective value %.10g.\n", objval); */
2297
2298    cur_numcols = CPXgetnumcols (env, lp);
2299    cur_numrows = CPXgetnumrows (env, lp);
2300
2301    /* Retrieve basis, if one is available */
2302
2303    if ( solntype == CPX_BASIC_SOLN ) {
2304      cstat = (int *) malloc (cur_numcols*sizeof(int));
2305      rstat = (int *) malloc (cur_numrows*sizeof(int));
2306      if ( cstat == NULL || rstat == NULL ) {
2307        fprintf (stderr, "No memory for basis statuses.\n");
2308        goto TERMINATE;
2309      }
2310
2311      status = CPXgetbase (env, lp, cstat, rstat);
2312      if ( status ) {
2313        fprintf (stderr, "Failed to get basis; error %d.\n", status);
2314        goto TERMINATE;
2315      }
2316    }
2317    else {
2318      printf ("No basis available\n");
2319    }
2320
2321    /* Retrieve solution vector */
2322    x = (double *) malloc (cur_numcols*sizeof(double));
2323    if ( x == NULL ) {
2324      fprintf (stderr, "No memory for solution.\n");
2325      goto TERMINATE;
2326    }
2327
2328    status = CPXgetx (env, lp, x, 0, cur_numcols-1);
2329    if ( status ) {
2330      fprintf (stderr, "Failed to obtain primal solution.\n");
2331      goto TERMINATE;
2332    }
2333
2334    congestion = x[0];
2335    ptr = 1;
2336    /* printf ("b_i=%d, b_j=%d\n", b_i, b_j); */
2337    for (p=0; p<node_num; ++p){
2338      for (q=0; q<node_num; ++q){
```

```
2339        for (i=0; i<node_num; i++){
2340          for (j=0; j<node_num; j++){
2341            routingmatrix[p][q][i][j] = x[ptr];
2342            ptr++;
2343          }
2344        }
2345      }
2346    }
2347
2348    /* Display the maximum bound violation. */
2349    /* status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS); */
2350    /* if ( status ) { */
2351    /*   fprintf (stderr, "Failed to obtain bound violation.\n"); */
2352    /*   goto TERMINATE; */
2353    /* } */
2354    /* printf ("Maximum bound violation = %17.10g\n", maxviol); */
2355
2356
2357   TERMINATE:
2358    free_and_null ((char **) &cstat);
2359    free_and_null ((char **) &rstat);
2360    free_and_null ((char **) &x);
2361
2362    free(rhs);
2363    free(lb);
2364    free(ub);
2365    free(obj);
2366    free(sense);
2367    free(rowlist);
2368    free(collist);
2369    free(vallist);
2370    for (i=0; i<row_num; ++i){
2371      free(rowname[i]);
2372    }
2373    free(rowname);
2374    for (i=0; i<col_num; ++i){
2375      free(colname[i]);
2376    }
2377    free(colname);
2378
2379    if ( lp != NULL ) {
2380      status = CPXfreeprob (env, &lp);
2381      if ( status ) {
2382        fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
2383      }
2384    }
2385
2386    /* Free up the CPLEX environment, if necessary */
2387
2388    if ( env != NULL ) {
2389      status = CPXcloseCPLEX (&env);
2390
2391      /* Note that CPXcloseCPLEX produces no output,
2392         so the only way to see the cause of the error is to use
2393         CPXgeterrorstring.  For other CPLEX routines, the errors will
2394         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
2395
2396      if ( status ) {
2397        char errmsg[CPXMESSAGEBUFSIZE];
2398        fprintf (stderr, "Could not close CPLEX environment.\n");
```

```
2399         CPXgeterrorstring (env, status, errmsg);
2400         fprintf (stderr, "%s", errmsg);
2401       }
2402     }
2403
2404     return (status < 0 ? (double)status : congestion);
2405     /* return status; */
2406   }
```

各スキームのネットワーク輻輳率の算出プログラム

```
#include "routing_scheme.h"

double
mpls_pipe_congestion (int            node_num,
                      unsigned int **adjacencymatrix,
                      unsigned int **capacitymatrix,
                      unsigned int **trafficmatrix)
{
  double       ****routingmatrix;
  double         **linktrafficmatrix;
  double         **utilizationmatrix;
  unsigned int   **linkcostmatrix;
  int              i,j,k;
  int              max_link_i, max_link_j;
  double           congestion = -1.0;

  if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
   == NULL) MEM_EXIT;
  if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
   == NULL) MEM_EXIT;
  if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
   == NULL) MEM_EXIT;
  if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
   == NULL) MEM_EXIT;
  for (i=0; i<node_num; ++i){
    if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
      == NULL) MEM_EXIT;
    memset(linkcostmatrix[i], 0, node_num);
    if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
      == NULL) MEM_EXIT;
    if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
      == NULL) MEM_EXIT;
    if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
      == NULL) MEM_EXIT;
    for (j=0; j<node_num; ++j){
      if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
     == NULL) MEM_EXIT;
      for (k=0; k<node_num; ++k){
        if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
     == NULL) MEM_EXIT;
      }
    }
  }

  make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
  /* MPLS */
  if(opt_mpls_pipe(node_num, capacitymatrix, trafficmatrix, routingmatrix) != 0){
    fprintf(stderr, "error on calculate mpls\n");
    goto TERMINATE;
  }
  cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
  cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
 &max_link_i, &max_link_j);
  congestion = utilizationmatrix[max_link_i][max_link_j];

 TERMINATE:
  for (i=0; i < node_num; ++i){
    for (j=0; j < node_num; ++j){
```

```
59        for (k=0; k < node_num; ++k){
60          free(routingmatrix[i][j][k]);
61          routingmatrix[i][j][k] = NULL;
62        }
63        free(routingmatrix[i][j]);
64        routingmatrix[i][j] = NULL;
65      }
66      free(linktrafficmatrix[i]);
67      linktrafficmatrix[i] = NULL;
68      free(utilizationmatrix[i]);
69      utilizationmatrix[i] = NULL;
70      free(linkcostmatrix[i]);
71      linkcostmatrix[i]    = NULL;
72      free(routingmatrix[i]);
73      routingmatrix[i]     = NULL;
74    }
75    free(linktrafficmatrix);
76    linktrafficmatrix = NULL;
77    free(utilizationmatrix);
78    utilizationmatrix = NULL;
79    free(linkcostmatrix);
80    linkcostmatrix    = NULL;
81    free(routingmatrix);
82    routingmatrix     = NULL;
83
84    return (congestion);
85
86  }
87
88  double
89  spr_pipe_congestion (int            node_num,
90                       unsigned int **adjacencymatrix,
91                       unsigned int **capacitymatrix,
92                       unsigned int **trafficmatrix)
93
94  {
95    double       ****routingmatrix;
96    double         **linktrafficmatrix;
97    double         **utilizationmatrix;
98    unsigned int   **linkcostmatrix;
99    unsigned int   **nexthopmatrix;
100   int              i,j,k;
101   int              max_link_i, max_link_j;
102   double           congestion = -1.0;
103
104
105   if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
106     == NULL) MEM_EXIT;
107   if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
108     == NULL) MEM_EXIT;
109   if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
110     == NULL) MEM_EXIT;
111   if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
112     == NULL) MEM_EXIT;
113   if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
114     == NULL) MEM_EXIT;
115   for (i=0; i<node_num; ++i){
116     if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
117     == NULL) MEM_EXIT;
118     memset(nexthopmatrix[i], 0, node_num);
```

```
119       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
120        == NULL) MEM_EXIT;
121       memset(linkcostmatrix[i], 0, node_num);
122       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
123        == NULL) MEM_EXIT;
124       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
125        == NULL) MEM_EXIT;
126       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
127        == NULL) MEM_EXIT;
128       for (j=0; j<node_num; ++j){
129         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
130          == NULL) MEM_EXIT;
131         for (k=0; k<node_num; ++k){
132           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
133           == NULL) MEM_EXIT;
134         }
135       }
136     }
137
138     make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
139     Dijkstra_wo_ancestor(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix);
140     make_ospf_routing_matrix(node_num, nexthopmatrix, routingmatrix);
141     cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
142     cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
143    &max_link_i, &max_link_j);
144     congestion = utilizationmatrix[max_link_i][max_link_j];
145
146     for (i=0; i < node_num; ++i){
147       for (j=0; j < node_num; ++j){
148         for (k=0; k < node_num; ++k){
149           free(routingmatrix[i][j][k]);
150           routingmatrix[i][j][k] = NULL;
151         }
152         free(routingmatrix[i][j]);
153         routingmatrix[i][j] = NULL;
154       }
155       free(nexthopmatrix[i]);
156       nexthopmatrix[i]     = NULL;
157       free(linktrafficmatrix[i]);
158       linktrafficmatrix[i] = NULL;
159       free(utilizationmatrix[i]);
160       utilizationmatrix[i] = NULL;
161       free(linkcostmatrix[i]);
162       linkcostmatrix[i]    = NULL;
163       free(routingmatrix[i]);
164       routingmatrix[i]     = NULL;
165     }
166     free(nexthopmatrix);
167     nexthopmatrix     = NULL;
168     free(linktrafficmatrix);
169     linktrafficmatrix = NULL;
170     free(utilizationmatrix);
171     utilizationmatrix = NULL;
172     free(linkcostmatrix);
173     linkcostmatrix    = NULL;
174     free(routingmatrix);
175     routingmatrix     = NULL;
176
177     return (congestion);
178   }
```

```
179
180 double
181 spr_hose_congestion (int            node_num,
182                      unsigned int **adjacencymatrix,
183                      unsigned int **capacitymatrix,
184                      unsigned int **trafficmatrix
185                      )
186 {
187   double       ****routingmatrix;
188   double         **linktrafficmatrix;
189   double         **utilizationmatrix;
190   unsigned int   **linkcostmatrix;
191   unsigned int   **nexthopmatrix;
192   int              i,j,k;
193   unsigned int    *alpha;
194   unsigned int    *beta;
195   double           congestion = -1.0;
196   double           traffic;
197
198   if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
199     == NULL) MEM_EXIT;
200   if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
201     == NULL) MEM_EXIT;
202   if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
203     == NULL) MEM_EXIT;
204   if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
205    == NULL) MEM_EXIT;
206   if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
207    == NULL) MEM_EXIT;
208   if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
209    == NULL) MEM_EXIT;
210   memset(alpha, 0, node_num);
211   if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
212    == NULL) MEM_EXIT;
213   memset(beta, 0, node_num);
214   for (i=0; i<node_num; ++i){
215     if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
216     == NULL) MEM_EXIT;
217     memset(nexthopmatrix[i], 0, node_num);
218     if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
219     == NULL) MEM_EXIT;
220     memset(linkcostmatrix[i], 0, node_num);
221     if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
222     == NULL) MEM_EXIT;
223     if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
224     == NULL) MEM_EXIT;
225     if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
226     == NULL) MEM_EXIT;
227     for (j=0; j<node_num; ++j){
228       if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
229       == NULL) MEM_EXIT;
230       for (k=0; k<node_num; ++k){
231         if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
232         == NULL) MEM_EXIT;
233       }
234     }
235   }
236
237   /* make alpha & beta */
238   for (i=0; i<node_num; ++i){
```

```
239       alpha[i] = beta[i] = 0;
240     }
241     for (i=0; i<node_num; ++i){
242       for (j=0; j<node_num; ++j){
243         alpha[i] += trafficmatrix[i][j];
244         beta[j] += trafficmatrix[i][j];
245       }
246     }
247
248
249     make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
250     Dijkstra_wo_ancestor(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix);
251     make_ospf_routing_matrix(node_num, nexthopmatrix, routingmatrix);
252     congestion = 0.0;
253     for (i=0; i<node_num; ++i){
254       for (j=0; j<node_num; ++j){
255         if (capacitymatrix[i][j] > 0){
256           if ((traffic = opt_traffic_flow_hose(node_num, i, j, alpha, beta, routingmatrix)) < 0){
257             fprintf(stderr, "error on calculate traffic in spr_hose\n");
258             congestion = -1.0;
259             goto TERMINATE;
260           }
261           if (congestion < (traffic/capacitymatrix[i][j])){
262             congestion = traffic/capacitymatrix[i][j];
263           }
264         }
265       }
266     }
267
268   TERMINATE:
269     for (i=0; i < node_num; ++i){
270       for (j=0; j < node_num; ++j){
271         for (k=0; k < node_num; ++k){
272           free(routingmatrix[i][j][k]);
273           routingmatrix[i][j][k] = NULL;
274         }
275         free(routingmatrix[i][j]);
276         routingmatrix[i][j] = NULL;
277       }
278       free(nexthopmatrix[i]);
279       nexthopmatrix[i]     = NULL;
280       free(linktrafficmatrix[i]);
281       linktrafficmatrix[i] = NULL;
282       free(utilizationmatrix[i]);
283       utilizationmatrix[i] = NULL;
284       free(linkcostmatrix[i]);
285       linkcostmatrix[i]    = NULL;
286       free(routingmatrix[i]);
287       routingmatrix[i]     = NULL;
288     }
289     free(nexthopmatrix);
290     nexthopmatrix     = NULL;
291     free(linktrafficmatrix);
292     linktrafficmatrix = NULL;
293     free(utilizationmatrix);
294     utilizationmatrix = NULL;
295     free(linkcostmatrix);
296     linkcostmatrix    = NULL;
297     free(routingmatrix);
298     routingmatrix     = NULL;
```

```
299
300     return (congestion);
301   }
302
303
304   double
305   sospf_split_pipe_congestion (int            node_num,
306                                unsigned int **adjacencymatrix,
307                                unsigned int **capacitymatrix,
308                                unsigned int **trafficmatrix)
309   {
310     double       ****routingmatrix;
311     double         **linktrafficmatrix;
312     double         **utilizationmatrix;
313     unsigned int   **linkcostmatrix;
314     unsigned int   **nexthopmatrix;
315     unsigned int  ***ancestormatrix;
316     int              i,j,k;
317     int              max_link_i, max_link_j;
318     double           congestion = -1.0;
319
320     if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
321     == NULL) MEM_EXIT;
322     if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
323     == NULL) MEM_EXIT;
324     if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
325     == NULL) MEM_EXIT;
326     if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
327     == NULL) MEM_EXIT;
328     if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
329     == NULL) MEM_EXIT;
330     if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
331     == NULL) MEM_EXIT;
332     for (i=0; i<node_num; ++i){
333       if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
334       == NULL) MEM_EXIT;
335       memset(nexthopmatrix[i], 0, node_num);
336       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
337       == NULL) MEM_EXIT;
338       memset(linkcostmatrix[i], 0, node_num);
339       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
340       == NULL) MEM_EXIT;
341       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
342       == NULL) MEM_EXIT;
343       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
344       == NULL) MEM_EXIT;
345       if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
346       == NULL) MEM_EXIT;
347       for (j=0; j<node_num; ++j){
348         if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
349          == NULL) MEM_EXIT;
350         memset(ancestormatrix[i][j], 0, node_num);
351         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
352          == NULL) MEM_EXIT;
353         for (k=0; k<node_num; ++k){
354           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
355           == NULL) MEM_EXIT;
356         }
357       }
358     }
```

```
359
360     make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
361     Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
362     if(opt_sospf_split_pipe(node_num, nexthopmatrix, ancestormatrix, capacitymatrix,
363    trafficmatrix, routingmatrix) != 0){
364       fprintf(stderr, "error on calculate sospf\n");
365       goto TERMINATE;
366     }
367
368     cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
369     cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
370    &max_link_i, &max_link_j);
371
372
373    TERMINATE:
374     for (i=0; i < node_num; ++i){
375       for (j=0; j < node_num; ++j){
376         for (k=0; k < node_num; ++k){
377           free(routingmatrix[i][j][k]);
378           routingmatrix[i][j][k] = NULL;
379         }
380         free(ancestormatrix[i][j]);
381         ancestormatrix[i][i] = NULL;
382         free(routingmatrix[i][j]);
383         routingmatrix[i][j] = NULL;
384       }
385       free(nexthopmatrix[i]);
386       nexthopmatrix[i]     = NULL;
387       free(linktrafficmatrix[i]);
388       linktrafficmatrix[i] = NULL;
389       free(utilizationmatrix[i]);
390       utilizationmatrix[i] = NULL;
391       free(linkcostmatrix[i]);
392       linkcostmatrix[i]    = NULL;
393       free(ancestormatrix[i]);
394       ancestormatrix[i]    = NULL;
395       free(routingmatrix[i]);
396       routingmatrix[i]     = NULL;
397     }
398     free(nexthopmatrix);
399     nexthopmatrix     = NULL;
400     free(linktrafficmatrix);
401     linktrafficmatrix = NULL;
402     free(utilizationmatrix);
403     utilizationmatrix = NULL;
404     free(linkcostmatrix);
405     linkcostmatrix    = NULL;
406     free(ancestormatrix);
407     ancestormatrix    = NULL;
408     free(routingmatrix);
409     routingmatrix     = NULL;
410
411     return (congestion);
412
413   }
414
415
416   double
417   spr_pipe_congestion_lf (int            node_num,
418                           unsigned int **adjacencymatrix,
```

```
419                         unsigned int **capacitymatrix,
420                         unsigned int **trafficmatrix,
421                         double       **congestmatrix)
422  {
423    double       ****routingmatrix;
424    double         **linktrafficmatrix;
425    double         **utilizationmatrix;
426    unsigned int   **linkcostmatrix;
427    unsigned int   **nexthopmatrix;
428    int              i,j,k,b_i,b_j,tmp_ci,tmp_cj;
429    int              max_link_i, max_link_j;
430
431    if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
432            == NULL) MEM_EXIT;
433    if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
434      == NULL) MEM_EXIT;
435    if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
436             == NULL) MEM_EXIT;
437    if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
438     == NULL) MEM_EXIT;
439    if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
440              == NULL) MEM_EXIT;
441    for (i=0; i<node_num; ++i){
442      if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
443   == NULL) MEM_EXIT;
444      memset(nexthopmatrix[i], 0, node_num);
445      if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
446    == NULL) MEM_EXIT;
447      memset(linkcostmatrix[i], 0, node_num);
448      if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
449           == NULL) MEM_EXIT;
450      if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
451           == NULL) MEM_EXIT;
452      if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
453              == NULL) MEM_EXIT;
454      for (j=0; j<node_num; ++j){
455        if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
456        == NULL) MEM_EXIT;
457        for (k=0; k<node_num; ++k){
458          if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
459      == NULL) MEM_EXIT;
460        }
461      }
462    }
463
464    for(b_i=0; b_i<node_num; ++b_i){
465      for(b_j=0; b_j<node_num; ++b_j){
466        congestmatrix[b_i][b_j] = 1000;
467      }
468    }
469
470
471    /* make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix); */
472    /* Dijkstra_wo_ancestor(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix); */
473    /* make_ospf_routing_matrix(node_num, nexthopmatrix, routingmatrix); */
474
475    /* /\*  *\/ */
476    /* cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix); */
477
478    for(b_i=0; b_i<node_num; ++b_i){
```

```
479       for(b_j=0; b_j<node_num; ++b_j){
480         if(b_i <= b_j && capacitymatrix[b_i][b_j] != 0){
481           tmp_ci = capacitymatrix[b_i][b_j];
482           tmp_cj = capacitymatrix[b_j][b_i];
483           capacitymatrix[b_i][b_j] = 0;
484           capacitymatrix[b_j][b_i] = 0;
485
486           make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
487           Dijkstra_wo_ancestor(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix);
488           make_ospf_routing_matrix(node_num, nexthopmatrix, routingmatrix);
489           cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
490           cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
491    &max_link_i, &max_link_j);
492
493           congestmatrix[b_i][b_j] = utilizationmatrix[max_link_i][max_link_j];
494
495           capacitymatrix[b_i][b_j] = tmp_ci;
496           capacitymatrix[b_j][b_i] = tmp_cj;
497         }
498       }
499     }
500
501
502
503     for (i=0; i < node_num; ++i){
504       for (j=0; j < node_num; ++j){
505         for (k=0; k < node_num; ++k){
506           free(routingmatrix[i][j][k]);
507           routingmatrix[i][j][k] = NULL;
508         }
509         free(routingmatrix[i][j]);
510         routingmatrix[i][j] = NULL;
511       }
512       free(nexthopmatrix[i]);
513       nexthopmatrix[i]     = NULL;
514       free(linktrafficmatrix[i]);
515       linktrafficmatrix[i] = NULL;
516       free(utilizationmatrix[i]);
517       utilizationmatrix[i] = NULL;
518       free(linkcostmatrix[i]);
519       linkcostmatrix[i]    = NULL;
520       free(routingmatrix[i]);
521       routingmatrix[i]     = NULL;
522     }
523     free(nexthopmatrix);
524     nexthopmatrix     = NULL;
525     free(linktrafficmatrix);
526     linktrafficmatrix = NULL;
527     free(utilizationmatrix);
528     utilizationmatrix = NULL;
529     free(linkcostmatrix);
530     linkcostmatrix    = NULL;
531     free(routingmatrix);
532     routingmatrix     = NULL;
533
534     return 0;
535   }
536
537   double
538   spr_hose_congestion_lf (int            node_num,
```

```
539                         unsigned int **adjacencymatrix,
540                         unsigned int **capacitymatrix,
541                         unsigned int **trafficmatrix,
542                         double       **congestmatrix)
543
544
545   {
546     double       ****routingmatrix;
547     double         **linktrafficmatrix;
548     double         **utilizationmatrix;
549     unsigned int   **linkcostmatrix;
550     unsigned int   **nexthopmatrix;
551     int              i,j,k,b_i,b_j,tmp_ci,tmp_cj;
552     unsigned int    *alpha;
553     unsigned int    *beta;
554     double           congestion = -1.0;
555     double           traffic;
556
557
558     if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
559       == NULL) MEM_EXIT;
560     if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
561       == NULL) MEM_EXIT;
562     if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
563             == NULL) MEM_EXIT;
564     if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
565      == NULL) MEM_EXIT;
566     if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
567      == NULL) MEM_EXIT;
568     if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
569        == NULL) MEM_EXIT;
570     memset(alpha, 0, node_num);
571     if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
572        == NULL) MEM_EXIT;
573     memset(beta, 0, node_num);
574     for (i=0; i<node_num; ++i){
575       if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
576    == NULL) MEM_EXIT;
577       memset(nexthopmatrix[i], 0, node_num);
578       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
579     == NULL) MEM_EXIT;
580       memset(linkcostmatrix[i], 0, node_num);
581       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
582            == NULL) MEM_EXIT;
583       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
584            == NULL) MEM_EXIT;
585       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
586               == NULL) MEM_EXIT;
587       for (j=0; j<node_num; ++j){
588         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
589         == NULL) MEM_EXIT;
590         for (k=0; k<node_num; ++k){
591           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
592       == NULL) MEM_EXIT;
593         }
594       }
595     }
596
597
598     for(b_i=0; b_i<node_num; ++b_i){
```

```
599       for(b_j=0; b_j<node_num; ++b_j){
600         congestmatrix[b_i][b_j] = 1000;
601       }
602     }
603
604
605     /* make alpha & beta */
606     for (i=0; i<node_num; ++i){
607       alpha[i] = beta[i] = 0;
608     }
609     for (i=0; i<node_num; ++i){
610       for (j=0; j<node_num; ++j){
611         alpha[i] += trafficmatrix[i][j];
612         beta[j] += trafficmatrix[i][j];
613       }
614     }
615
616
617     for(b_i=0; b_i<node_num; ++b_i){
618       for(b_j=0; b_j<node_num; ++b_j){
619         if(b_i <= b_j && capacitymatrix[b_i][b_j] != 0){
620           tmp_ci = capacitymatrix[b_i][b_j];
621           tmp_cj = capacitymatrix[b_j][b_i];
622           capacitymatrix[b_i][b_j] = 0;
623           capacitymatrix[b_j][b_i] = 0;
624           make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
625           Dijkstra_wo_ancestor(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix);
626           make_ospf_routing_matrix(node_num, nexthopmatrix, routingmatrix);
627           congestion = 0.0;
628           for (i=0; i<node_num; ++i){
629             for (j=0; j<node_num; ++j){
630               if (capacitymatrix[i][j] > 0){
631                 if ((traffic = opt_traffic_flow_hose(node_num, i, j, alpha, beta, routingmatrix))
632     < 0){
633                   fprintf(stderr, "error on calculate traffic in spr_hose\n");
634                   congestion = -1.0;
635                   goto TERMINATE;
636                 }
637                 if (congestion < (traffic/capacitymatrix[i][j])){
638                   congestion = traffic/capacitymatrix[i][j];
639                 }
640               }
641             }
642           }
643
644           congestmatrix[b_i][b_j] = congestion;
645
646           capacitymatrix[b_i][b_j] = tmp_ci;
647           capacitymatrix[b_j][b_i] = tmp_cj;
648         }
649       }
650     }
651
652
653     TERMINATE:
654     for (i=0; i < node_num; ++i){
655       for (j=0; j < node_num; ++j){
656         for (k=0; k < node_num; ++k){
657           free(routingmatrix[i][j][k]);
658           routingmatrix[i][j][k] = NULL;
```

```
659         }
660         free(routingmatrix[i][j]);
661         routingmatrix[i][j] = NULL;
662       }
663       free(nexthopmatrix[i]);
664       nexthopmatrix[i]     = NULL;
665       free(linktrafficmatrix[i]);
666       linktrafficmatrix[i] = NULL;
667       free(utilizationmatrix[i]);
668       utilizationmatrix[i] = NULL;
669       free(linkcostmatrix[i]);
670       linkcostmatrix[i]    = NULL;
671       free(routingmatrix[i]);
672       routingmatrix[i]     = NULL;
673     }
674     free(nexthopmatrix);
675     nexthopmatrix     = NULL;
676     free(linktrafficmatrix);
677     linktrafficmatrix = NULL;
678     free(utilizationmatrix);
679     utilizationmatrix = NULL;
680     free(linkcostmatrix);
681     linkcostmatrix    = NULL;
682     free(routingmatrix);
683     routingmatrix     = NULL;
684
685     return 0;
686   }
687
688   double
689   sospf_split_hose_congestion (int            node_num,
690                                unsigned int **adjacencymatrix,
691                                unsigned int **capacitymatrix,
692                                unsigned int **trafficmatrix
693                                )
694   {
695     double       ****routingmatrix;
696     double         **linktrafficmatrix;
697     double         **utilizationmatrix;
698     unsigned int   **linkcostmatrix;
699     unsigned int   **nexthopmatrix;
700     unsigned int  ***ancestormatrix;
701     unsigned int    *alpha;
702     unsigned int    *beta;
703     int              i,j,k;
704     int              max_link_i, max_link_j;
705     double           congestion = -1.0;
706
707
708     if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
709                 == NULL) MEM_EXIT;
710     if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
711          == NULL) MEM_EXIT;
712     if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
713                  == NULL) MEM_EXIT;
714     if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
715          == NULL) MEM_EXIT;
716     if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
717                   == NULL) MEM_EXIT;
718     if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
```

```
719         == NULL) MEM_EXIT;
720     if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
721         == NULL) MEM_EXIT;
722     memset(alpha, 0, node_num);
723     if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
724         == NULL) MEM_EXIT;
725     memset(beta, 0, node_num);
726     for (i=0; i<node_num; ++i){
727       if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
728        == NULL) MEM_EXIT;
729       memset(nexthopmatrix[i], 0, node_num);
730       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
731         == NULL) MEM_EXIT;
732       memset(linkcostmatrix[i], 0, node_num);
733       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
734                == NULL) MEM_EXIT;
735       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
736                == NULL) MEM_EXIT;
737       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
738                == NULL) MEM_EXIT;
739       if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
740       == NULL) MEM_EXIT;
741       for (j=0; j<node_num; ++j){
742         if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
743     == NULL) MEM_EXIT;
744         memset(ancestormatrix[i][j], 0, node_num);
745         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
746             == NULL) MEM_EXIT;
747         for (k=0; k<node_num; ++k){
748           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
749           == NULL) MEM_EXIT;
750         }
751       }
752     }
753
754     /* make alpha & beta */
755     for (i=0; i<node_num; ++i){
756       alpha[i] = beta[i] = 0;
757     }
758     for (i=0; i<node_num; ++i){
759       for (j=0; j<node_num; ++j){
760         alpha[i] += trafficmatrix[i][j];
761         beta[j] += trafficmatrix[i][j];
762       }
763     }
764
765     make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
766     Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
767     /* SOSPF */
768     if((congestion = opt_sospf_split_hose(node_num, nexthopmatrix, ancestormatrix,
769     capacitymatrix, alpha, beta, routingmatrix)) < 0){
770       fprintf(stderr, "error on calculate sospf\n");
771       goto TERMINATE;
772     }
773     printf( "S-OSPF_h : %.20f\n", congestion);
774
775     TERMINATE:
776     for (i=0; i < node_num; ++i){
777       for (j=0; j < node_num; ++j){
778         for (k=0; k < node_num; ++k){
```

```
779           free(routingmatrix[i][j][k]);
780           routingmatrix[i][j][k] = NULL;
781         }
782         free(ancestormatrix[i][j]);
783         ancestormatrix[i][i] = NULL;
784         free(routingmatrix[i][j]);
785         routingmatrix[i][j] = NULL;
786       }
787       free(nexthopmatrix[i]);
788       nexthopmatrix[i]     = NULL;
789       free(linktrafficmatrix[i]);
790       linktrafficmatrix[i] = NULL;
791       free(utilizationmatrix[i]);
792       utilizationmatrix[i] = NULL;
793       free(linkcostmatrix[i]);
794       linkcostmatrix[i]    = NULL;
795       free(ancestormatrix[i]);
796       ancestormatrix[i]    = NULL;
797       free(routingmatrix[i]);
798       routingmatrix[i]     = NULL;
799     }
800     free(nexthopmatrix);
801     nexthopmatrix     = NULL;
802     free(linktrafficmatrix);
803     linktrafficmatrix = NULL;
804     free(utilizationmatrix);
805     utilizationmatrix = NULL;
806     free(linkcostmatrix);
807     linkcostmatrix    = NULL;
808     free(ancestormatrix);
809     ancestormatrix    = NULL;
810     free(routingmatrix);
811     routingmatrix     = NULL;
812     free(alpha);
813     alpha = NULL;
814     free(beta);
815     beta = NULL;
816
817     return (congestion);
818
819   }
820
821   double
822   sospf_pipe_local_reroute (int            node_num,
823                             unsigned int **adjacencymatrix,
824                             unsigned int **capacitymatrix,
825                             unsigned int **trafficmatrix,
826                             double       **congestmatrix)
827   {
828     double       ****routingmatrix;
829     double       ****backupmatrix;
830     double         **linktrafficmatrix;
831     double         **utilizationmatrix;
832     unsigned int   **linkcostmatrix;
833     unsigned int   **nexthopmatrix;
834     unsigned int   **newnexthop;
835     unsigned int  ***ancestormatrix;
836     int              p,q,i,j,k,b_i,b_j,b_num;
837     int              max_link_i, max_link_j, tmp_ci, tmp_cj;
838     int              status = 0;
```

```
839     double           congestion;
840
841     if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
842                 == NULL) MEM_EXIT;
843     if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
844           == NULL) MEM_EXIT;
845     if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
846                  == NULL) MEM_EXIT;
847     if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
848          == NULL) MEM_EXIT;
849     if((newnexthop   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
850          == NULL) MEM_EXIT;
851     if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
852                   == NULL) MEM_EXIT;
853     if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
854          == NULL) MEM_EXIT;
855     if((backupmatrix = (double ****)malloc(node_num * sizeof(double ***)))
856                   == NULL) MEM_EXIT;
857     for (i=0; i<node_num; ++i){
858       if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
859        == NULL) MEM_EXIT;
860       memset(nexthopmatrix[i], 0, node_num);
861       if((newnexthop[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
862        == NULL) MEM_EXIT;
863       memset(newnexthop[i], 0, node_num);
864       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
865         == NULL) MEM_EXIT;
866       memset(linkcostmatrix[i], 0, node_num);
867       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
868                 == NULL) MEM_EXIT;
869       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
870                 == NULL) MEM_EXIT;
871       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
872                 == NULL) MEM_EXIT;
873       if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
874       == NULL) MEM_EXIT;
875       if((backupmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
876                 == NULL) MEM_EXIT;
877       for (j=0; j<node_num; ++j){
878         if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
879    == NULL) MEM_EXIT;
880         memset(ancestormatrix[i][j], 0, node_num);
881         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
882             == NULL) MEM_EXIT;
883         if((backupmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
884              == NULL) MEM_EXIT;
885         for (k=0; k<node_num; ++k){
886           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
887           == NULL) MEM_EXIT;
888           if((backupmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
889            == NULL) MEM_EXIT;
890         }
891       }
892     }
893
894     for(b_i=0; b_i<node_num; ++b_i){
895       for(b_j=0; b_j<node_num; ++b_j){
896         congestmatrix[b_i][b_j] = 1000;
897       }
898     }
```

```
899     make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
900     Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
901     if(opt_sospf_split_pipe(node_num, nexthopmatrix, ancestormatrix, capacitymatrix,
902   trafficmatrix, routingmatrix) != 0){
903       fprintf(stderr, "error on calculate sospf\n");
904       goto TERMINATE;
905     }
906
907     for(p=0; p<node_num; ++p){
908       for(q=0; q<node_num; ++q){
909         for(i=0; i<node_num; ++i){
910           for(j=0; j<node_num; ++j){
911             backupmatrix[p][q][i][j] = routingmatrix[p][q][i][j];
912           }
913         }
914       }
915     }
916
917     for(b_i=0; b_i<node_num; ++b_i){
918       for(b_j=b_i+1; b_j<node_num; ++b_j){
919         if(capacitymatrix[b_i][b_j] != 0){
920           tmp_ci = capacitymatrix[b_i][b_j];
921           tmp_cj = capacitymatrix[b_j][b_i];
922           capacitymatrix[b_i][b_j] = 0;
923           capacitymatrix[b_j][b_i] = 0;
924
925           b_num = cal_break_num(routingmatrix, node_num, b_i, b_j);
926           make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
927           Dijkstra(node_num, adjacencymatrix, linkcostmatrix, newnexthop, ancestormatrix);
928
929           if(opt_sospf_local_reroute_pipe(node_num, b_num, b_i, b_j, newnexthop, ancestormatrix,
930   capacitymatrix, trafficmatrix, routingmatrix) != 0){
931             fprintf(stderr, "error on calculate local\n");
932             status = 1;
933             goto TERMINATE;
934           }
935           cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
936           cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
937   &max_link_i, &max_link_j);
938           if(congestmatrix[b_i][b_j] > utilizationmatrix[max_link_i][max_link_j]){
939             congestmatrix[b_i][b_j] = utilizationmatrix[max_link_i][max_link_j];
940           }
941           capacitymatrix[b_i][b_j] = tmp_ci;
942           capacitymatrix[b_j][b_i] = tmp_cj;
943           for(p=0; p<node_num; ++p){
944             for(q=0; q<node_num; ++q){
945               for(i=0; i<node_num; ++i){
946                 for(j=0; j<node_num; ++j){
947                   routingmatrix[p][q][i][j] = backupmatrix[p][q][i][j];
948                 }
949               }
950             }
951           }
952         }
953       }
954     }
955
956   TERMINATE:
957     for (i=0; i < node_num; ++i){
958       for (j=0; j < node_num; ++j){
```

```
959         for (k=0; k < node_num; ++k){
960           free(routingmatrix[i][j][k]);
961           routingmatrix[i][j][k] = NULL;
962           free(backupmatrix[i][j][k]);
963           backupmatrix[i][j][k] = NULL;
964         }
965         free(ancestormatrix[i][j]);
966         ancestormatrix[i][i] = NULL;
967         free(routingmatrix[i][j]);
968         routingmatrix[i][j] = NULL;
969         free(backupmatrix[i][j]);
970         backupmatrix[i][j] = NULL;
971       }
972       free(nexthopmatrix[i]);
973       nexthopmatrix[i]     = NULL;
974       free(newnexthop[i]);
975       newnexthop[i]        = NULL;
976       free(linktrafficmatrix[i]);
977       linktrafficmatrix[i] = NULL;
978       free(utilizationmatrix[i]);
979       utilizationmatrix[i] = NULL;
980       free(linkcostmatrix[i]);
981       linkcostmatrix[i]    = NULL;
982       free(ancestormatrix[i]);
983       ancestormatrix[i]    = NULL;
984       free(routingmatrix[i]);
985       routingmatrix[i]     = NULL;
986       free(backupmatrix[i]);
987       backupmatrix[i]     = NULL;
988     }
989     free(nexthopmatrix);
990     nexthopmatrix     = NULL;
991     free(linktrafficmatrix);
992     linktrafficmatrix = NULL;
993     free(utilizationmatrix);
994     utilizationmatrix = NULL;
995     free(linkcostmatrix);
996     linkcostmatrix    = NULL;
997     free(ancestormatrix);
998     ancestormatrix    = NULL;
999     free(routingmatrix);
1000    routingmatrix     = NULL;
1001    free(backupmatrix);
1002    backupmatrix     = NULL;
1003
1004    return 0;
1005
1006  }
1007
1008  double
1009  sospf_hose_local_reroute (int            node_num,
1010                            unsigned int **adjacencymatrix,
1011                            unsigned int **capacitymatrix,
1012                            unsigned int **trafficmatrix,
1013                            double       **congestmatrix)
1014  {
1015    double       ****routingmatrix;
1016    double       ****backupmatrix;
1017    double         **linktrafficmatrix;
1018    double         **utilizationmatrix;
```

```
1019    unsigned int   **linkcostmatrix;
1020    unsigned int   **nexthopmatrix;
1021    unsigned int   **newnexthop;
1022    unsigned int  ***ancestormatrix;
1023    unsigned int    *alpha;
1024    unsigned int    *beta;
1025    int              p,q,i,j,k,b_i,b_j,b_num;
1026    int              max_link_i, max_link_j, tmp_ci, tmp_cj;
1027    int              status = 0;
1028    double           congestion;
1029
1030
1031    if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
1032                == NULL) MEM_EXIT;
1033    if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1034          == NULL) MEM_EXIT;
1035    if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
1036                 == NULL) MEM_EXIT;
1037    if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1038         == NULL) MEM_EXIT;
1039    if((newnexthop   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1040         == NULL) MEM_EXIT;
1041    if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
1042                  == NULL) MEM_EXIT;
1043    if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
1044         == NULL) MEM_EXIT;
1045    if((backupmatrix = (double ****)malloc(node_num * sizeof(double ***)))
1046                  == NULL) MEM_EXIT;
1047    if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1048         == NULL) MEM_EXIT;
1049    memset(alpha, 0, node_num);
1050    if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1051         == NULL) MEM_EXIT;
1052    memset(beta, 0, node_num);
1053    for (i=0; i<node_num; ++i){
1054      if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1055       == NULL) MEM_EXIT;
1056      memset(nexthopmatrix[i], 0, node_num);
1057      if((newnexthop[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1058       == NULL) MEM_EXIT;
1059      memset(newnexthop[i], 0, node_num);
1060      if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1061        == NULL) MEM_EXIT;
1062      memset(linkcostmatrix[i], 0, node_num);
1063      if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1064                == NULL) MEM_EXIT;
1065      if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1066                == NULL) MEM_EXIT;
1067      if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
1068                == NULL) MEM_EXIT;
1069      if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
1070      == NULL) MEM_EXIT;
1071      if((backupmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
1072              == NULL) MEM_EXIT;
1073      for (j=0; j<node_num; ++j){
1074        if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1075   == NULL) MEM_EXIT;
1076        memset(ancestormatrix[i][j], 0, node_num);
1077        if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
1078            == NULL) MEM_EXIT;
```

```
1079        if((backupmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
1080             == NULL) MEM_EXIT;
1081        for (k=0; k<node_num; ++k){
1082          if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
1083         == NULL) MEM_EXIT;
1084          if((backupmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
1085           == NULL) MEM_EXIT;
1086        }
1087      }
1088    }
1089
1090    for(b_i=0; b_i<node_num; ++b_i){
1091      for(b_j=0; b_j<node_num; ++b_j){
1092        congestmatrix[b_i][b_j] = 1000;
1093      }
1094    }
1095
1096    /* make alpha & beta */
1097    for (i=0; i<node_num; ++i){
1098      alpha[i] = beta[i] = 0;
1099    }
1100    for (i=0; i<node_num; ++i){
1101      for (j=0; j<node_num; ++j){
1102        alpha[i] += trafficmatrix[i][j];
1103        beta[j] += trafficmatrix[i][j];
1104      }
1105    }
1106
1107    make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
1108    Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
1109    if(opt_sospf_split_hose(node_num, nexthopmatrix, ancestormatrix, capacitymatrix,
1110   alpha, beta, routingmatrix) < 0){
1111      fprintf(stderr, "error on calculate sospf\n");
1112      goto TERMINATE;
1113    }
1114
1115    for(p=0; p<node_num; ++p){
1116      for(q=0; q<node_num; ++q){
1117        for(i=0; i<node_num; ++i){
1118          for(j=0; j<node_num; ++j){
1119            backupmatrix[p][q][i][j] = routingmatrix[p][q][i][j];
1120          }
1121        }
1122      }
1123    }
1124
1125    for(b_i=0; b_i<node_num; ++b_i){
1126      for(b_j=b_i+1; b_j<node_num; ++b_j){
1127        if(capacitymatrix[b_i][b_j] != 0){
1128          tmp_ci = capacitymatrix[b_i][b_j];
1129          tmp_cj = capacitymatrix[b_j][b_i];
1130          capacitymatrix[b_i][b_j] = 0;
1131          capacitymatrix[b_j][b_i] = 0;
1132
1133          b_num = cal_break_num(routingmatrix, node_num, b_i, b_j);
1134          make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
1135          Dijkstra(node_num, adjacencymatrix, linkcostmatrix, newnexthop, ancestormatrix);
1136
1137          if((congestion = opt_sospf_local_reroute_hose(node_num, b_num, b_i, b_j, newnexthop,
1138   ancestormatrix, capacitymatrix, trafficmatrix, routingmatrix)) < 0){
```

```
1139            fprintf(stderr, "error on calculate local_hose\n");
1140            goto TERMINATE;
1141          }
1142          congestmatrix[b_i][b_j] = congestion;
1143          capacitymatrix[b_i][b_j] = tmp_ci;
1144          capacitymatrix[b_j][b_i] = tmp_cj;
1145          for(p=0; p<node_num; ++p){
1146            for(q=0; q<node_num; ++q){
1147              for(i=0; i<node_num; ++i){
1148                for(j=0; j<node_num; ++j){
1149                  routingmatrix[p][q][i][j] = backupmatrix[p][q][i][j];
1150                }
1151              }
1152            }
1153          }
1154        }
1155      }
1156    }
1157
1158    TERMINATE:
1159     for (i=0; i < node_num; ++i){
1160       for (j=0; j < node_num; ++j){
1161         for (k=0; k < node_num; ++k){
1162           free(routingmatrix[i][j][k]);
1163           routingmatrix[i][j][k] = NULL;
1164           free(backupmatrix[i][j][k]);
1165           backupmatrix[i][j][k] = NULL;
1166         }
1167         free(ancestormatrix[i][j]);
1168         ancestormatrix[i][i] = NULL;
1169         free(routingmatrix[i][j]);
1170         routingmatrix[i][j] = NULL;
1171         free(backupmatrix[i][j]);
1172         backupmatrix[i][j] = NULL;
1173       }
1174       free(nexthopmatrix[i]);
1175       nexthopmatrix[i]     = NULL;
1176       free(newnexthop[i]);
1177       newnexthop[i]        = NULL;
1178       free(linktrafficmatrix[i]);
1179       linktrafficmatrix[i] = NULL;
1180       free(utilizationmatrix[i]);
1181       utilizationmatrix[i] = NULL;
1182       free(linkcostmatrix[i]);
1183       linkcostmatrix[i]    = NULL;
1184       free(ancestormatrix[i]);
1185       ancestormatrix[i]    = NULL;
1186       free(routingmatrix[i]);
1187       routingmatrix[i]     = NULL;
1188       free(backupmatrix[i]);
1189       backupmatrix[i]     = NULL;
1190     }
1191     free(nexthopmatrix);
1192     nexthopmatrix     = NULL;
1193     free(linktrafficmatrix);
1194     linktrafficmatrix = NULL;
1195     free(utilizationmatrix);
1196     utilizationmatrix = NULL;
1197     free(linkcostmatrix);
1198     linkcostmatrix    = NULL;
```

```
1199    free(ancestormatrix);
1200    ancestormatrix    = NULL;
1201    free(routingmatrix);
1202    routingmatrix     = NULL;
1203    free(backupmatrix);
1204    backupmatrix      = NULL;
1205
1206    return 0;
1207
1208  }
1209
1210
1211  double
1212  sospf_pipe_congestion_all_reroute (int           node_num,
1213                                     unsigned int **adjacencymatrix,
1214                                     unsigned int **capacitymatrix,
1215                                     unsigned int **trafficmatrix,
1216                                     double       **congestmatrix)
1217  {
1218    double       ****routingmatrix;
1219    double         **linktrafficmatrix;
1220    double         **utilizationmatrix;
1221    unsigned int   **linkcostmatrix;
1222    unsigned int   **nexthopmatrix;
1223    unsigned int  ***ancestormatrix;
1224    int              i,j,k,b_i,b_j, tmp_ci, tmp_cj;
1225    int              max_link_i, max_link_j;
1226    double           congestion = -1.0;
1227
1228    if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
1229               == NULL) MEM_EXIT;
1230    if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1231         == NULL) MEM_EXIT;
1232    if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
1233                == NULL) MEM_EXIT;
1234    if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1235         == NULL) MEM_EXIT;
1236    if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
1237                 == NULL) MEM_EXIT;
1238    if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
1239         == NULL) MEM_EXIT;
1240    for (i=0; i<node_num; ++i){
1241      if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1242      == NULL) MEM_EXIT;
1243      memset(nexthopmatrix[i], 0, node_num);
1244      if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1245        == NULL) MEM_EXIT;
1246      memset(linkcostmatrix[i], 0, node_num);
1247      if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1248               == NULL) MEM_EXIT;
1249      if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1250               == NULL) MEM_EXIT;
1251      if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
1252               == NULL) MEM_EXIT;
1253      if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
1254      == NULL) MEM_EXIT;
1255      for (j=0; j<node_num; ++j){
1256        if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1257   == NULL) MEM_EXIT;
1258        memset(ancestormatrix[i][j], 0, node_num);
```

```
1259        if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
1260            == NULL) MEM_EXIT;
1261        for (k=0; k<node_num; ++k){
1262          if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
1263          == NULL) MEM_EXIT;
1264        }
1265      }
1266    }
1267
1268
1269    for(b_i=0; b_i<node_num; ++b_i){
1270      for(b_j=0; b_j<node_num; ++b_j){
1271        congestmatrix[b_i][b_j] = 1000;
1272      }
1273    }
1274
1275    for(b_i=0; b_i<node_num; ++b_i){
1276      for(b_j=b_i+1; b_j<node_num; ++b_j){
1277        if(capacitymatrix[b_i][b_j] != 0){
1278          tmp_ci = capacitymatrix[b_i][b_j];
1279          tmp_cj = capacitymatrix[b_j][b_i];
1280          capacitymatrix[b_i][b_j] = 0;
1281          capacitymatrix[b_j][b_i] = 0;
1282
1283          make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
1284          Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
1285          if(opt_sospf_split_pipe(node_num, nexthopmatrix, ancestormatrix, capacitymatrix,
1286    trafficmatrix, routingmatrix) != 0){
1287            fprintf(stderr, "error on calculate sospf\n");
1288            goto TERMINATE;
1289          }
1290          cal_link_traffic(node_num, trafficmatrix, routingmatrix, linktrafficmatrix);
1291          cal_link_utilization(node_num, capacitymatrix, linktrafficmatrix, utilizationmatrix,
1292    &max_link_i, &max_link_j);
1293          congestmatrix[b_i][b_j] = utilizationmatrix[max_link_i][max_link_j];
1294
1295          capacitymatrix[b_i][b_j] = tmp_ci;
1296          capacitymatrix[b_j][b_i] = tmp_cj;
1297        }
1298      }
1299    }
1300
1301    TERMINATE:
1302    for (i=0; i < node_num; ++i){
1303      for (j=0; j < node_num; ++j){
1304        for (k=0; k < node_num; ++k){
1305          free(routingmatrix[i][j][k]);
1306          routingmatrix[i][j][k] = NULL;
1307        }
1308        free(ancestormatrix[i][j]);
1309        ancestormatrix[i][i] = NULL;
1310        free(routingmatrix[i][j]);
1311        routingmatrix[i][j] = NULL;
1312      }
1313      free(nexthopmatrix[i]);
1314      nexthopmatrix[i]     = NULL;
1315      free(linktrafficmatrix[i]);
1316      linktrafficmatrix[i] = NULL;
1317      free(utilizationmatrix[i]);
1318      utilizationmatrix[i] = NULL;
```

```
1319      free(linkcostmatrix[i]);
1320      linkcostmatrix[i]    = NULL;
1321      free(ancestormatrix[i]);
1322      ancestormatrix[i]    = NULL;
1323      free(routingmatrix[i]);
1324      routingmatrix[i]     = NULL;
1325    }
1326    free(nexthopmatrix);
1327    nexthopmatrix     = NULL;
1328    free(linktrafficmatrix);
1329    linktrafficmatrix = NULL;
1330    free(utilizationmatrix);
1331    utilizationmatrix = NULL;
1332    free(linkcostmatrix);
1333    linkcostmatrix    = NULL;
1334    free(ancestormatrix);
1335    ancestormatrix    = NULL;
1336    free(routingmatrix);
1337    routingmatrix     = NULL;
1338
1339    return 0;
1340
1341  }
1342
1343
1344  double
1345  sospf_hose_congestion_all_reroute (int            node_num,
1346                                     unsigned int **adjacencymatrix,
1347                                     unsigned int **capacitymatrix,
1348                                     unsigned int **trafficmatrix,
1349                                     double       **congestmatrix)
1350  {
1351    double       ****routingmatrix;
1352    double         **linktrafficmatrix;
1353    double         **utilizationmatrix;
1354    unsigned int   **linkcostmatrix;
1355    unsigned int   **nexthopmatrix;
1356    unsigned int  ***ancestormatrix;
1357    unsigned int    *alpha;
1358    unsigned int    *beta;
1359    int              i,j,k,b_i,b_j, tmp_ci, tmp_cj;
1360    int              max_link_i, max_link_j;
1361    double           congestion = -1.0;
1362
1363
1364    if((linktrafficmatrix   = (double **)malloc(node_num * sizeof(double *)))
1365                == NULL) MEM_EXIT;
1366    if((linkcostmatrix = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1367         == NULL) MEM_EXIT;
1368    if((utilizationmatrix   = (double **)malloc(node_num * sizeof(double*)))
1369                 == NULL) MEM_EXIT;
1370    if((nexthopmatrix   = (unsigned int **)malloc(node_num * sizeof(unsigned int*)))
1371         == NULL) MEM_EXIT;
1372    if((routingmatrix = (double ****)malloc(node_num * sizeof(double ***)))
1373                  == NULL) MEM_EXIT;
1374    if((ancestormatrix = (unsigned int***)malloc(node_num * sizeof(unsigned int**)))
1375         == NULL) MEM_EXIT;
1376    if((alpha = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1377         == NULL) MEM_EXIT;
1378    memset(alpha, 0, node_num);
```

```
1379     if((beta = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1380          == NULL) MEM_EXIT;
1381     memset(beta, 0, node_num);
1382     for (i=0; i<node_num; ++i){
1383       if((nexthopmatrix[i]   = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1384        == NULL) MEM_EXIT;
1385       memset(nexthopmatrix[i], 0, node_num);
1386       if((linkcostmatrix[i] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1387         == NULL) MEM_EXIT;
1388       memset(linkcostmatrix[i], 0, node_num);
1389       if((linktrafficmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1390                == NULL) MEM_EXIT;
1391       if((utilizationmatrix[i]   = (double*)malloc(node_num * sizeof(double)))
1392                == NULL) MEM_EXIT;
1393       if((routingmatrix[i] = (double ***)malloc(node_num * sizeof(double **)))
1394                == NULL) MEM_EXIT;
1395       if((ancestormatrix[i] = (unsigned int**)malloc(node_num * sizeof(unsigned int*)))
1396       == NULL) MEM_EXIT;
1397       for (j=0; j<node_num; ++j){
1398         if((ancestormatrix[i][j] = (unsigned int*)malloc(node_num * sizeof(unsigned int)))
1399    == NULL) MEM_EXIT;
1400         memset(ancestormatrix[i][j], 0, node_num);
1401         if((routingmatrix[i][j] = (double **)malloc(node_num * sizeof(double *)))
1402             == NULL) MEM_EXIT;
1403         for (k=0; k<node_num; ++k){
1404           if((routingmatrix[i][j][k] = (double *)malloc(node_num * sizeof(double)))
1405           == NULL) MEM_EXIT;
1406         }
1407       }
1408     }
1409
1410     for(b_i=0; b_i<node_num; ++b_i){
1411       for(b_j=0; b_j<node_num; ++b_j){
1412         congestmatrix[b_i][b_j] = 1000;
1413       }
1414     }
1415
1416     /* make alpha & beta */
1417     for (i=0; i<node_num; ++i){
1418       alpha[i] = beta[i] = 0;
1419     }
1420     for (i=0; i<node_num; ++i){
1421       for (j=0; j<node_num; ++j){
1422         alpha[i] += trafficmatrix[i][j];
1423         beta[j] += trafficmatrix[i][j];
1424       }
1425     }
1426     for(b_i=0; b_i<node_num; ++b_i){
1427       for(b_j=b_i+1; b_j<node_num; ++b_j){
1428         if(capacitymatrix[b_i][b_j] != 0){
1429           tmp_ci = capacitymatrix[b_i][b_j];
1430           tmp_cj = capacitymatrix[b_j][b_i];
1431           capacitymatrix[b_i][b_j] = 0;
1432           capacitymatrix[b_j][b_i] = 0;
1433
1434           make_linkcost_matrix_based_capacity(node_num, capacitymatrix, linkcostmatrix);
1435           Dijkstra(node_num, adjacencymatrix, linkcostmatrix, nexthopmatrix, ancestormatrix);
1436           /* SOSPF */
1437           if((congestion = opt_sospf_split_hose(node_num, nexthopmatrix, ancestormatrix,
1438    capacitymatrix, alpha, beta, routingmatrix)) < 0){
```

```
1439            fprintf(stderr, "error on calculate sospf\n");
1440            goto TERMINATE;
1441          }
1442          congestmatrix[b_i][b_j] = congestion;
1443          capacitymatrix[b_i][b_j] = tmp_ci;
1444          capacitymatrix[b_j][b_i] = tmp_cj;
1445        }
1446      }
1447    }
1448
1449   TERMINATE:
1450    for (i=0; i < node_num; ++i){
1451      for (j=0; j < node_num; ++j){
1452        for (k=0; k < node_num; ++k){
1453          free(routingmatrix[i][j][k]);
1454          routingmatrix[i][j][k] = NULL;
1455        }
1456        free(ancestormatrix[i][j]);
1457        ancestormatrix[i][i] = NULL;
1458        free(routingmatrix[i][j]);
1459        routingmatrix[i][j] = NULL;
1460      }
1461      free(nexthopmatrix[i]);
1462      nexthopmatrix[i]     = NULL;
1463      free(linktrafficmatrix[i]);
1464      linktrafficmatrix[i] = NULL;
1465      free(utilizationmatrix[i]);
1466      utilizationmatrix[i] = NULL;
1467      free(linkcostmatrix[i]);
1468      linkcostmatrix[i]    = NULL;
1469      free(ancestormatrix[i]);
1470      ancestormatrix[i]    = NULL;
1471      free(routingmatrix[i]);
1472      routingmatrix[i]     = NULL;
1473    }
1474    free(nexthopmatrix);
1475    nexthopmatrix     = NULL;
1476    free(linktrafficmatrix);
1477    linktrafficmatrix = NULL;
1478    free(utilizationmatrix);
1479    utilizationmatrix = NULL;
1480    free(linkcostmatrix);
1481    linkcostmatrix    = NULL;
1482    free(ancestormatrix);
1483    ancestormatrix    = NULL;
1484    free(routingmatrix);
1485    routingmatrix     = NULL;
1486    free(alpha);
1487    alpha = NULL;
1488    free(beta);
1489    beta = NULL;
1490
1491    return 0;
1492
1493  }
```

```
#include "routing_tools.h"

int Dijkstra(int node_num,
             unsigned int **adjacencymatrix,
             unsigned int **linkcostmatrix,
             unsigned int **nexthopmatrix,
             unsigned int ***ancestormatrix)
{

  int           past_hop[node_num][node_num];
  unsigned long sum_cost[node_num][node_num];
  unsigned int  routeflg[node_num];
  unsigned int  source, destination, target, min_cost_node, i, j, k, ptr, flg_sum, next;
  unsigned long min_cost;

  for (i=0; i<node_num; ++i){
    for (j=0; j<node_num; ++j){
      for (k=0; k<node_num; ++k){
        nexthopmatrix[j][k] = 0;
        ancestormatrix[i][j][k] = 0;
      }
    }
  }

  for (source=0; source < node_num; ++source){
    for (i=0; i<node_num; ++i){
      routeflg[i] = 0;
      for (j=0; j<node_num; ++j){
        past_hop[i][j] = -1;
        sum_cost[i][j] = ULONG_MAX;
      }
    }

    past_hop[source][source] = source;
    sum_cost[source][source] = 0;
    routeflg[source]         = 1;
    ptr                      = source;

    do{
      min_cost = ULONG_MAX;
      min_cost_node = source;
      for (i=0; i<node_num; ++i){
        if(routeflg[i]             != 1 &&
           adjacencymatrix[ptr][i] == 1) {
          if(sum_cost[source][i] >
             (sum_cost[source][ptr] + linkcostmatrix[ptr][i])){

            sum_cost[source][i] = sum_cost[source][ptr] + linkcostmatrix[ptr][i];
            past_hop[source][i] = ptr;
          }
        }
      }

      for (i=0; i<node_num; ++i){
        if((sum_cost[source][i] < min_cost) && (routeflg[i] != 1)){
          min_cost      = sum_cost[source][i];
          min_cost_node = i;
        }
```

```
59        }
60
61        ptr = min_cost_node;
62        routeflg[ptr] = 1;
63        for (i=0, flg_sum=0; i<node_num; ++i){
64          flg_sum += routeflg[i];
65        }
66      } while (flg_sum < node_num);
67
68      for (i=0; i<node_num; ++i){
69        if(i == source){
70          nexthopmatrix[i][source] = source;
71        }else{
72          next = i;
73          while((j = past_hop[source][next]) != source){
74            next = j;
75          }
76          nexthopmatrix[i][source] = next;
77        }
78      }
79    }
80
81    for (source=0; source<node_num; ++source){
82      for(destination=0; destination<node_num; ++destination){
83        for(target=0; target<node_num; ++target){
84          if(adjacencymatrix[source][target] == 1){
85            next = target;
86            while((next = nexthopmatrix[destination][next]) != destination){
87              if (next == source){
88                ancestormatrix[destination][source][target] = 1;
89                break;
90              }
91            }
92          }
93        }
94      }
95    }
96    return 0;
97  }
98
99  int Dijkstra_wo_ancestor(int            node_num,
100                           unsigned int **adjacencymatrix,
101                           unsigned int **linkcostmatrix,
102                           unsigned int **nexthopmatrix)
103 {
104
105   int           past_hop[node_num][node_num];
106   unsigned long sum_cost[node_num][node_num];
107   unsigned int  routeflg[node_num];
108   unsigned int  source, min_cost_node, i, j, ptr, flg_sum, next;
109   unsigned long min_cost;
110
111   for (i=0; i<node_num; ++i){
112     for (j=0; j<node_num; ++j){
113       nexthopmatrix[i][j] = 0;
114     }
115   }
116
117   for (source=0; source < node_num; ++source){
118     for (i=0; i<node_num; ++i){
```

```
119         routeflg[i] = 0;
120         for (j=0; j<node_num; ++j){
121           past_hop[i][j] = -1;
122           sum_cost[i][j] = ULONG_MAX;
123         }
124       }
125
126       past_hop[source][source] = source;
127       sum_cost[source][source] = 0;
128       routeflg[source]         = 1;
129       ptr                      = source;
130
131       do{
132         min_cost                      = ULONG_MAX;
133         min_cost_node                 = source;
134         for (i=0; i<node_num; ++i){
135           if(routeflg[i]             != 1 &&
136              adjacencymatrix[ptr][i] == 1) {
137             if(sum_cost[source][i] >
138                (sum_cost[source][ptr] + linkcostmatrix[ptr][i])){
139               sum_cost[source][i]     = sum_cost[source][ptr] + linkcostmatrix[ptr][i];
140               past_hop[source][i]     = ptr;
141             }
142           }
143         }
144
145         for (i=0; i<node_num; ++i){
146           if((sum_cost[source][i] < min_cost) && (routeflg[i] != 1)){
147             min_cost      = sum_cost[source][i];
148             min_cost_node = i;
149           }
150         }
151
152         ptr = min_cost_node;
153         routeflg[ptr] = 1;
154         for (i=0, flg_sum=0; i<node_num; ++i){
155           flg_sum += routeflg[i];
156         }
157       } while (flg_sum < node_num);
158       for (i=0; i<node_num; ++i){
159         if(i == source){
160           nexthopmatrix[i][source] = source;
161         }else{
162           next = i;
163           while((j = past_hop[source][next]) != source){
164             next = j;
165           }
166           nexthopmatrix[i][source] = next;
167         }
168       }
169     }
170
171     return 0;
172   }
173
174   int
175   make_ospf_routing_matrix (int              node_num,
176                             unsigned int   **nexthopmatrix,
177                             double       ****routingmatrix)
178   {
```

```
179     int p,q,current,i,j;
180
181     for (p=0; p<node_num ; ++p){
182       for (q=0; q<node_num; ++q){
183         for (i=0; i<node_num; ++i){
184           for (j=0; j<node_num; ++j){
185             routingmatrix[p][q][i][j] = 0;
186           }
187         }
188       }
189     }
190
191     for (p=0; p<node_num; ++p){
192       for (q=0; q<node_num; ++q){
193         current = p;
194         while(1){
195           routingmatrix[p][q][current][nexthopmatrix[q][current]] = 1;
196           if ((current = nexthopmatrix[q][current]) == q){
197             break;
198           }
199         }
200       }
201     }
202     return 0;
203   }
204
205   int
206   ospf_backup_routing (int              node_num,
207                        unsigned int   **oldnext,
208                        unsigned int   **newnext,
209                        double       ****routematrix)
210   {
211     int p,q,i,j,k;
212     double tmp;
213     int current;
214     double sum;
215     int ****check;
216
217     if((check = (int ****)malloc(node_num * sizeof(int ***)))         == NULL);
218     for (i=0; i<node_num; ++i){
219       if((check[i] = (int ***)malloc(node_num * sizeof(int **)))      == NULL);
220       for (j=0; j<node_num; ++j){
221         if((check[i][j] = (int **)malloc(node_num * sizeof(int *)))   == NULL);
222         for (k=0; k<node_num; ++k){
223           if((check[i][j][k] = (int *)malloc(node_num * sizeof(int))) == NULL);
224         }
225       }
226     }
227
228    for (p=0; p<node_num; ++p){
229       for (q=0; q<node_num; ++q){
230         for (i=0; i<node_num; ++i){
231           for (j=0; j<node_num; ++j){
232            check[p][q][i][j] = 0;
233           }
234         }
235       }
236     }
237
238     for (p=0; p<node_num; ++p){
```

```
239       for (q=0; q<node_num; ++q){
240         if(p != q){
241           sum = 0;
242           current = 0;
243           i = p;
244           check[p][q][i][current] = 0;
245           while(current<node_num){
246             if(routematrix[p][q][i][current] != 0){
247               check[p][q][i][current] = 1;
248               tmp = routematrix[p][q][i][current];
249               sum += routematrix[p][q][i][current];
250               i = current;
251               j = oldnext[q][i];
252               while(1){
253                 if((oldnext[q][i] != p) && (routematrix[p][q][i][j] != 0)){
254                   if(check[p][q][i][j] == 1 && tmp <= 1){
255                     tmp += routematrix[p][q][i][j];
256                   }
257                   routematrix[p][q][i][j] = tmp;
258                   check[p][q][i][j] = 1;
259                   i = j;
260                   j = oldnext[q][i];
261                 }
262                 else if((oldnext[q][i] == p) || (routematrix[p][q][i][oldnext[q][i]] == 0)){
263                   while(1){
264                     check[p][q][i][newnext[q][i]] = 1;
265                     routematrix[p][q][i][newnext[q][i]] += tmp;
266                     if ((i = newnext[q][i]) == q){
267                       break;
268                     }
269                   }
270                 }
271                 if(i == q){
272                   i = p;
273                   break;
274                 }
275               }
276             }
277             if(sum==1){
278               break;
279             }
280             else{
281               current++;
282             }
283           }
284         }
285       }
286     }
287
288     for (i=0; i < node_num; ++i){
289       for (j=0; j < node_num; ++j){
290         for (k=0; k < node_num; ++k){
291           free(check[i][j][k]);
292           check[i][j][k] = NULL;
293         }
294         free(check[i][j]);
295         check[i][j] = NULL;
296       }
297       free(check[i]);
298     }
```

```
299     free(check);
300     check     = NULL;
301
302     return 0;
303   }
304
305   int
306   cal_link_traffic (int              node_num,
307                     unsigned int   **trafficmatrix,
308                     double       ****routingmatrix,
309                     double         **linktrafficmatrix)
310   {
311     int i,j,p,q;
312
313     for (i=0; i<node_num; ++i){
314       for (j=0; j<node_num; ++j){
315         linktrafficmatrix[i][j] = 0;
316       }
317     }
318
319     for (p=0; p<node_num; ++p){
320       for (q=0; q<node_num; ++q){
321         for (i=0; i<node_num; ++i){
322           for (j=0; j<node_num; ++j){
323             linktrafficmatrix[i][j] += trafficmatrix[p][q] * routingmatrix[p][q][i][j];
324           }
325         }
326       }
327     }
328     return 0;
329   }
330
331   int
332   cal_link_utilization(int            node_num,
333                        unsigned int **capacitymatrix,
334                        double       **linktrafficmatrix,
335                        double       **utilizationmatrix,
336                        int           *link_i,
337                        int           *link_j)
338   {
339     int i,j;
340     double max = 0;
341
342     for (i=0; i<node_num; ++i){
343       for (j=0; j<node_num; ++j){
344         utilizationmatrix[i][j] = 0;
345       }
346     }
347
348     for (i=0; i<node_num; ++i){
349       for (j=0; j<node_num; ++j){
350         if (capacitymatrix[i][j] != 0){
351           utilizationmatrix[i][j] = linktrafficmatrix[i][j] / (double)capacitymatrix[i][j];
352           if(max < utilizationmatrix[i][j]){
353             max = utilizationmatrix[i][j];
354             *link_i = i;
355             *link_j = j;
356           }
357         }else{
358           utilizationmatrix[i][j] = -1.0;
```

```
359         }
360       }
361     }
362     return 0;
363   }
364
365   int
366   normalize_matrix(int               node_num,
367                    int               division,
368                    double           **routingmatrix)
369   {
370     int i,j;
371     for(i=0; i<node_num; ++i){
372       for(j=0; j<node_num; ++j){
373         routingmatrix[i][j] = (double)routingmatrix[i][j] / (double)division;
374       }
375     }
376     return 0;
377   }
378
379   int
380   link_failure(int            node_num,
381                int            b_i,
382                int            b_j,
383                unsigned int **nexthopmatrix,
384                double       ****routingmatrix)
385
386   {
387     int    p, q, i, j;
388     double tmp_x;
389     int    erase;
390
391
392     for(p=0; p<node_num; ++p){
393       for(q=0; q<node_num; ++q){
394         erase = 0;
395         if(routingmatrix[p][q][b_i][b_j] != 0){
396           tmp_x = routingmatrix[p][q][b_i][b_j];
397           i = b_i;
398           j = b_j;
399           while(erase != 1){
400             routingmatrix[p][q][i][j] = routingmatrix[p][q][i][j] - tmp_x;
401             i = j;
402             j = nexthopmatrix[q][i];
403             if(i == q){
404               erase = 1;
405             }
406           }
407         }
408         else if(routingmatrix[p][q][b_j][b_i] != 0){
409           tmp_x = routingmatrix[p][q][b_j][b_i];
410           i = b_j;
411           j = b_i;
412           while(erase != 1){
413             routingmatrix[p][q][i][j] = routingmatrix[p][q][i][j] - tmp_x;
414             i = j;
415             j = nexthopmatrix[q][i];
416             if(i == q){
417               erase = 1;
418             }
```

```
419           }
420         }
421       }
422     }
423
424     return 0;
425
426   }
427
```