UNIVERSIDADE DOS AÇORES

Departamento de Economia e Gestão

Modelos Regret aplicados a Problemas de Localização

Dissertação de Doutoramento

Pedro Portugal de Sousa Nunes

2012

Universidade dos Açores
Departamento de Economia e Gestão

Modelos Regret aplicados a Problemas de Localização

Pedro Portugal de Sousa Nunes

Dissertação submetida à Universidade dos Açores no âmbito das provas para a obtenção do grau de Doutor em Ciências Económicas e Empresariais, na especialidade de Planeamento Económico e Métodos Quantitativos, orientada pela Professora Doutora Helena Ramalhinho Dias Lourenço, Professora Associada do Departamento de Economia e Gestão da Universidade Pompeu e Fabra e Coorientação do Professor Doutor Francisco José Silva, Professor Auxiliar do Departamento de Economia e Gestão da Universidade dos Açores.

2012

Agradecimentos

Gostava de expressar o meu agradecimento aos meus orientadores, Professora Doutora Helena Ramalhinho Dias Lourenço e Professor Doutor Francisco José Silva, por todo o apoio que me foi prestado e sugestões que me foram apresentadas.

A todos os que se interessaram, que me motivaram e apoiaram, direta ou indiretamente. Este trabalho não teria sido possível sem a V. presença.

Resumo

Dada a importância da prestação de serviços face à procura existente e aos custos inerentes à sua configuração, os problemas de localização são de suma importância quer no quotidiano quer no meio científico. Na tentativa de captar as particularidades desses sistemas e fazer uma aproximação à realidade, os modelos de localização tornaram-se de tal forma complexos que os resultados por enumeração completa se tornam de difícil obtenção fruto essencialmente do crescimento exponencial do tempo de computação.

O presente trabalho apresenta um algoritmo que, além do conhecido GRASP, incorpora o método *p-minmax* Regret com o objetivo de avaliar a solução heurística obtida no que concerne a sua robustez para diferentes cenários. A utilização desses processos vem na linha de pesquisas anteriores e visa a sua integração com o intuito de explorar novas metodologias que melhorem ou melhor se adaptem às circunstâncias dos casos estudados.

Fazendo variar os limites em termos de tempos de espera e de distância máxima percorrida, limites de capacidades de processamento da procura e dimensão das redes, é possível verificar mudanças significativas nas soluções finais. Os problemas em estudo são o bem conhecido Problema de Localização com Cobertura Máxima de ReVelle e um modelo alternativo no qual o comportamento de escolha do servidor não depende apenas do tempo percorrido do nó ao centro, mas também inclui o tempo de espera pelo serviço. Foram também abordados o Problema de Localização de Captura Máxima, do mesmo autor, e o Problema de Localização de Infraestruturas com Capacidades Limitadas com base no problema estudado originalmente por Balinski.

Os modelos testados e os seus vários exemplos foram obtidos com recurso à geração numérica aleatória. Em muitos casos, evidenciam-se resultados distintos mas existem outros onde a formulação proposta não produz diferenças significativas nos resultados. De uma forma geral,

nos sistemas mais “apertados”, ou seja, onde o limite de distância seja mais pequeno, o número de centros de serviço sejam em menor número ou as capacidades de processamento das infraestruturas menores, as decisões de localização são mais sensíveis aos parâmetros pré-definidos para o modelo.

Em conclusão, tendo-se simulado as populações e as respetivas frequências de procura, com este trabalho consegue-se evidenciar a suma importância, tal como na vida real, de considerar o congestionamento dos sistemas nas suas várias vertentes como um fator determinante nas decisões de localização e afetação.

Palavras chave: *localização, afetação, heurística, regret, filas de espera*.

Abstract

Given the importance of service delivery compared to existing demand and the costs associated with setting, location problems are of paramount importance both for daily issues and for the scientific community. In an attempt to capture the particularities of these systems and make an approximation to reality, the location models have become so complex that complete enumeration results become difficult to obtain essentially as a result of the exponential growth of computing time.

This paper presents an algorithm that, besides the known GRASP, incorporates the p-minmax Regret method in order to evaluate the heuristic solution obtained with respect to its robustness through different scenarios. The use of these processes is in line with previous research and aims at their integration in order to explore new methodologies that improve or best suit the circumstances of the cases studied.

By varying the limits in terms of waiting times and maximum distance traveled, maximum demand processing capabilities and size of the network, you can see significant changes in the final solutions. The problems under study are the well-known Location Problem with Maximum Coverage of ReVelle and an alternative model in which the server´s choice behavior not only depends on the elapsed time from the node to the center, but also includes the waiting time for the service. We also discussed the Maximum Capture Location Problem, by the same author, and the Limited Capacity Infrastructure Location Problem based on the problem originally studied by Balinski.

The models and their various examples were obtained using the random number generation. In many cases, different results are apparent but there are others in which the formulation proposed produces no significant differences in results. Generally, in systems more "tight", that is, where

the distance limit is smaller, the service centers are fewer in number or infrastructure´s processing capabilities smaller, location decisions are more sensitive to preset parameters for the model.

In conclusion, having been simulated populations and the respective frequencies of demand, this work manages to highlight the utmost importance, as in real life, considering the congestion of the systems in its various aspects as a determining factor in location and affectation decisions.

Keywords: *location*, *affectation*, *heuristic*, *regret*, *queues*.

Índice

Lista de Pseudo-Códigos

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Há cerca de 25 anos que questões relacionadas com a localização de infraestruturas prestadoras de serviços são o tema de estudo de vários investigadores. Exemplos destes serviços são os sistemas médicos, operações policiais, bombeiros, serviços de assistência na estrada, entre outros. Qualquer formulação dos referidos sistemas terá em comum a seguinte característica: o desempenho do serviço é na sua maioria definido pelo tempo que o cliente espera pelo serviço.

Nos últimos anos da década de 70 e também da década de 80 foi reforçado o interesse nos modelos de localização. Os diferentes modelos desenvolvidos têm em comum o facto de a sua complexidade dificultar o processo de encontrar uma solução. Desta forma, as formulações foram restringidas através de hipóteses simplificadoras, ocorrendo por vezes não estarem de acordo com a realidade enfrentada pelo planeador, quer no setor público ou privado. Os avanços tecnológicos permitiram um desenvolver gradual de formulações mais realistas, dada a possibilidade de encontrar uma solução para modelos complexos com tempos de computação aceitáveis.

Muitas formulações foram propostas para os vários problemas relacionados com a localização e afetação. Os modelos podem ser formulados e resolvidos recorrendo à programação linear, programação inteira, programação dinâmica ou aproximações heurísticas e meta-heurísticas. As instalações podem apresentar restrições em termos de capacidade, estarem sujeitas a congestionamento, terem níveis mínimos de serviço e outras características próprias. Todos estes fatores e variáveis conjugados com as novas tecnologias de processamento de dados, torna possível a modelação destas realidades e a respetiva procura por soluções viáveis no que respeitam os diversos objetivos propostos.

As regiões têm sido representadas por redes, um espaço contínuo ou um conjunto de pontos discretos. Em qualquer um dos casos os objetivos podem variar desde cobertura a captura, no que concerne modelos de maximização, ou custos no que diz respeito a modelos de minimização. Quando nos referirmos a "cobertura" pretende-se que determinada fração ou totalidade da procura seja satisfeita na solução encontrada enquanto que "captura" visa a localização competitiva com objetivo virado para a quota de mercado obtida. Por outro lado, podemos ter o interesse em decidir localizações de serviços que permitam obter soluções de custo mínimo sendo que aqui, além dos custos fixos relacionados com as próprias instalações, podemos associar adicionalmente um custo relacionado com a deslocação.

Porém, uma classificação geral de modelos de localização poderá ser considerada pelos seguintes grupos: modelos de distância máxima, probemas de "*p-dispersion*" e modelos de distância média ou total. Ver como exemplo Current et al. [1].

Os modelos de distância máxima consideram explicitamente uma distância máxima dentro da qual devem localizar uma instalação de forma que providencie o serviço. Este é geralmente o caso de quando se localizam escolas, hospitais ou estações de polícia em que as pessoas esperam ter uma instalação disponível dentro de limites aceitáveis de distância a partir da sua área de residência. Dois tipos de modelos foram desenvolvidos de acordo com estas categorias: modelos de cobertura e modelos "*p-center*".

No que diz respeito aos modelos de cobertura, é predefinido um valor máximo quer para a distância quer para o tempo de deslocação. Caso um serviço seja prestado por uma instalação localizada abaixo desse máximo, então o serviço é considerado adequado ou aceitável. Um consumidor é considerado coberto pelo servidor caso tenha uma instalação dentro do limite de distância pré-estabelecido. O modelo "*p-center*" de Hakimi [2] [3] considera o problema de

minimizar a distância máxima desde um nó de procura à instalação servidora mais próxima sendo que estamos a localizar um número de instalações conhecidas à *priori*.

Já tendo sido oportunamente referido, os modelos podem ser classificados tendo em conta o objetivo, mas também podem ser categorizados de acordo com o tipo de servidor (fixo ou móvel) e com o facto de este ter ou não associado uma capacidade no que concerne o volume de procura que poderá satisfazer (modelos com capacidade e sem capacidade).

O problema que os investigadores se propõem a resolver prende-se com a localização dos centros de serviço e com a respetiva afetação de procura a esses centros. Geralmente a performance desse tipo de serviços é avaliada pelo número de clientes na fila de espera e pelo tempo que a estes esperam desde que chegam ao centro. O que se poderá concluir é que estes indicadores estão muito correlacionados com o número de centros de prestação de serviços e a sua localização.

Marianov e Serra [4] introduzem o "*Queuing Maximal Covering Location Allocation Model*" que localiza p centros e afeta a este utilizadores de forma a maximizar a população coberta pelo serviço, sendo esta cobertura definida como (i) afeto a um centro de serviço num certo período de tempo ou distância de casa, e (ii) caso um cliente chegue ao respetivo centro de serviço, ele(a) será servido(a) de acordo com um tempo τ de chegada ao centro, com uma probabilidade de pelo menos α.

Silva e Serra [5], com base no "*Maximum Coverage Model*" e recorrendo a Filas de Espera com prioridades, desenvolvem o tema da localização de serviços de emergência incluindo prioridades nas chamadas. A complexidade adicional implica a utilização de uma heurística. O trabalho é desenvolvido com duas perspetivas: afetação forçada por entidade reguladora e afetação pela escolha do utilizador do serviço.

O presente trabalho desenvolve: no Capítulo 4 um modelo que herda a sua formulação de anteriores estudos do "*Maximum Coverage Models*" incorporando adicionalmente resultados da Teoria de Filas de Espera; no Capítulo 5 é estudado um modelo aplicado ao problema originalmente desenvolvido por Hotelling [6], com uma estrutura base análoga à desenvolvida por Revelle [7] no que concerne o "T*he Maximum Capture Availability*"; por último, no Capítulo 6, é trabalhado um modelo aplicado ao caso "*Facility Location Problem"* abordado originalmente por Balinski [8].

A complexidade associada ao modelo, assumida sofisticação na tentativa de captar mais elementos das realidades em estudo, obriga o recurso a métodos heurísticos na procura das soluções. Assim sendo, além do mencionado, o algoritmo contém uma componente *Regret*, baseada no trabalho de Daskin [9], que demonstra produzir resultados aceitáveis tanto em termos de velocidade de computação como, e porventura mais importante, em termos de aproximação à solução ótima.

Capítulo 2

Literatura Relacionada

A revisão bibliográfica efetuada tem como objeto de análise modelos de localização e afetação que adicionalmente incorporem efeitos de filas de espera e a respetiva simulação de modo a testar os parâmetros e prerrogativas do estudo.

Este capítulo desenvolve-se em 5 subcapítulos pretendendo agrupar as obras no que respeita os modelos estudados em termos de tipologia de modelo de localização. Desta forma começámos com os Modelos Discretos de Localização (seção 2.1) para, posteriormente, analisar os seus homólogos Modelos Estocásticos de Localização (seção 2.2). No que respeita os Modelos de Localização com Filas de Espera, optou-se por subdividi-los de acordo com a definição Servidores Móveis (seção 2.3) nos casos em que se trate, por exemplo, de ambulâncias ou serviços do género, e Servidores Fixos (seção 2.4) para, por exemplo, o caso de hospitais e centros de saúde. Por fim, com a inclusão da simulação na revisão levada a cabo, observa-se o caso dos Modelos de Localização com Simulação (seção 2.5).

2.1. Modelos Discretos de Localização.

Os Modelos Discretos de Localização têm sido objeto de estudo já há algumas décadas, tendo sido propostos tanto no setor público como no privado. No que diz respeito ao setor público, os modelos têm em linha de conta providenciar com eficiência serviços públicos enquanto no setor privado, a habilidade de uma empresa competir no mercado é a preocupação adicional. A maior parte das aplicações verificadas no setor público estão relacionadas com emergências

hospitalares, escolas, portos, aeroportos e serviços de administração pública. No setor privado podemos encontrar diversas aplicações no que concerne a localização de lojas a retalho, bem como instalações da indústria transformadora e armazéns.

Os Modelos Discretos de Localização são de um modo geral formulados com programações lineares inteiras, que podem ser solucionados recorrendo a algoritmo conhecidos tal como "*branch and bound*". No entanto, até o mais básico dos problemas de localização é classificado como "*NP-Hard*" e requereria um tempo de computação inaceitável para encontrar padrões de afetação associados a situações mais realistas.

Uma das primeiras referências aos Modelos de Localização resulta da formulação efetuada por Hotelling [6] na qual duas empresas competem num mercado linear. Hotelling considera o caso de uma procura inelástica e custos de produção unitários em ordem a determinar, simultaneamente, as localizações, os preços e os outputs de duas empresas idênticas localizadas em linha como o objetivo comum de maximizar os respetivos proveitos. Esta formulação tem sido referida como o primeiro modelo de localização competitiva, sendo maioritariamente aplicada no setor privado, onde os objetivos, em geral, estão ligados à maximização de proveitos.

Friesz, Miller e Tobin [10], definem um modelo de localização competitiva como qualquer modelo que reconheça explicitamente que a localização poderá afetar a quota de mercado sendo assim a escolha feita de forma a que os objetivos da empresa sejam otimizados no que diz respeito a essa quota de mercado.

ReVelle [11] reviu a formulação de Hotelling e desenvolveu um modelo de localização competitiva para uma rede: "*The Maximum Capture Problem*" (MAXCAP). O modelo

MAXCAP assume, entre outras, que a decisão do consumidor ao escolher a instalação servidora baseia-se na distância. O modelo MAXCAP considera que o mercado já tem localizados estabelecimentos pertencentes à empresa concorrente (empresa B) sendo que a nossa decisão de localização (A) será captar o máximo de procura com um número pré-determinado de servidores.

Outro estudo pioneiro foi o realizado por Hakimi [2, 3]. Hakimi considerou uma rede sem direção imposta aos arcos, estando os consumidores localizados apenas nos nós, com determinada fração de procura do consumidor retribuída de cada nó. O problema 1-mediano explorado baseia-se na localização numa rede de uma instalação que irá servir o consumidor de forma a minimizar a distância média de deslocação entre a instalação e a população de consumidores. Hakimi mostrou que, de entre o conjunto de nós, existia pelo menos uma localização ótima para a instalação servidora apresentada, reduzindo-se desta forma uma pesquisa contínua para uma simplesmente finita. Um resultado análogo é também aplicado ao problema multi-medianos, no qual diversas instalações devem ser localizadas de forma a minimizar a distância média percorrida desde os consumidores à instalação mais próxima.

O primeiro modelo de localização a considerar a distância máxima foi desenvolvido por Toregas (Toregas et al.[12]). Neste o objetivo é o de minimizar o número de instalações a serem localizadas de forma a cobrir toda a procura (bem como uma instalação dentro do tempo limite). A formulação resultante é a que se apresenta:

$$Min \quad Z = \sum_{j \in J} X_j \qquad (2.1)$$

$$sujeito \quad a$$

$$\sum_{j \in N_i} X_j \geq 1 \qquad N_i = \{j | t_{ji} < S\} \qquad (2.2)$$

$$X_j \in \{0,1\}$$

sendo,

$$X_j = \begin{cases} 1, \text{caso um servidor esteja localizado em j} \\ 0, \text{caso contrário} \end{cases}$$

A restrição (2.2) obriga a que cada nó de procura tenha pelo menos uma instalação dentro do tempo limite S. Para cada nó de procura uma vizinhança com raio S e a restrição forçam a que a soma de X_j dentro do raio seja maior ou igual a 1.

O modelo de cobertura máxima de Church e ReVelle [13], limita o número de instalações a serem localizadas a um outro limite: o objetivo é localizar um número pré-determinado de instalações de forma a maximizar a procura coberta. Este modelo não obriga a que toda a procura seja coberta. Por outro lado, ele procura localizar um número fixo de instalações que muito provavelmente não cobrirão toda a população.

No problema "*p-dispersion*" de Kuby [14], o objetivo vai na direção oposta do problema de p-centro, isto é, procura maximizar a distância entre qualquer par de instalações. O problema tem sido aplicado à localização de instalações indesejáveis e para na localização de pontos de franchise, onde a separação reduz o risco de competição nos mesmos mercados.

Uma formulação deste tipo de modelos é dada por

$$\begin{aligned} &Max \quad \mathrm{D} \\ &\text{sujeito a} \\ &\sum_{j \in J} \mathrm{X_j} = p && (2.3) \\ &D + (M - d_{ij}) X_i + (M - d_{ij}) X_j \leq 2M - d_{ij} \quad \forall \mathrm{i, j} \in \mathrm{J, i < j} && (2.4) \\ &\mathrm{X_j} \in \{0,1\} \quad \forall \mathrm{j} \in \mathrm{J} \end{aligned}$$

M é uma constante grande e D a distância de separação mínima entre qualquer par de instalações.

A função objetivo maximiza a distância entre as instalações mais próximas. A primeira restrição (2.3) requer que p instalações sejam localizadas. A segunda restrição (2.4) define a separação mínima entre qualquer par de instalações abertas. Se quer X_i ou X_j for zero, a restrição não é vinculativa. Se ambos forem iguais a um então a restrição é equivalente a $D \leq d_{ij}$. Assim sendo, maximizando *D* tem o efeito de forçar a menor distância entre a facilidade a ser tão grande quanto possível.

O terceiro grupo de modelos de localização, Modelos de Distância Médio ou Total inclui os problemas relacionados com a distância de viagem entre as instalações e os nós de procura. O problema p-mediano, (Hakimi, [2] [3]), encontra as localizações para um determinado número de instalações, de forma a minimizar a distância ponderada da procura total entre os nós de procura e as instalações às quais são atribuídas. Este modelo pode ser formulado como se apresenta:

$$Min \quad \sum_{i \in I} \sum_{j \in J} a_i d_{ij} Y_{ij} \qquad (2.5)$$

sujeito a

$$\sum_{j \in J} X_j = p \qquad (2.6)$$

$$\sum_{j \in J} Y_{ij} = 1 \qquad \forall i \in I \qquad (2.7)$$

$$Y_{ij} - X_j \leq 0 \qquad \forall i \in I, \forall j \in J \qquad (2.8)$$

$$X_j \in \{0,1\}, Y_{ij} \in \{0,1\} \qquad \forall i \in I, \forall j \in J$$

O objetivo minimiza a distância ponderada total percorrida pela população. A primeira restrição (2.6) fixa o número de instalações a serem localizadas. A segunda restrição (2.7) força cada ponto de procura a ser atribuído apenas um centro. A terceira restrição (2.8) força os pontos de procura a serem afetos apenas a instalações abertas. A formulação p-mediano tem sido amplamente utilizada em diferentes aplicações.

O problema "*Single Source Capacitated Plant Location*" verifica uma estrutura p-mediano mas incorpora outras caraterísticas ao considerar diferentes custos fixos para diferentes locais potenciais e assumir uma restrição de capacidade para as instalações que estão sendo localizadas. Este problema pretende minimizar o custo total da instalação e os custos de transporte.

Dois outros tipos de problemas podem ser incluídos no terceiro grupo de modelos de localização: o problema "*Hub Location*", que tenta localizar os centros e definir as rotas de entrega que tentam minimizar o custo total (que é uma função da distância) e o problema "*Maxisum Location*". Este último visa a localização de um determinado número dessas instalações de forma que a distância total ponderada entre os nós e as instalações prestadoras do serviço às quais estão atribuídas é maximizada.

2.2. Modelos Estocásticos de Localização.

Alguns problemas de localização de instalações enfrentam os problemas da incerteza na procura, tempo de deslocação ou custos de instalações. Desde os anos setenta, problemas relacionados com incerteza têm sido desenvolvidos na literatura sobre localização. Quatro abordagens básicas foram formuladas: aproximações através de um substituto determinista, encontrar equivalentes determinísticos, modelos probabilísticos com constrangimento, sistemas de filas espacialmente distribuídas e planeamento de cenários.

Os modelos determinísticos que abordam o congestionamento têm sido formulados como modelos de localização e cobertura com cobertura redundante. Uma possível formulação

matemática, baseada nos modelos BACOP 1 e BACOP 2 de Hogan e ReVelle [15], é apresentada a seguir:

BACOP 1:

$$Max \qquad Z = \sum_{i \in I} a_i r_i \qquad (2.9)$$

$$sujeito \quad a$$

$$r_i \leq \sum_{j \in N_i} X_j - 1 \qquad \forall i \in I \qquad (2.10)$$

$$\sum_{j \in J} X_j = p \qquad (2.11)$$

$$X_j, Y_i \in \{0,1\} \qquad \forall j \in J, i \in I$$

a_i é um peso caraterizador da importância de prever o ponto de procura i com servidores redundantes;

r_i é uma variável inteira que mede o número de servidores maior do que a unidade para o ponto de procura i.

O modelo BACOP 1 procura maximizar a população coberta com pelo menos um servidor adicional em relação ao primeiro.

BACOP 2:

$$Max \qquad Z_1 = \sum_{i \in I} a_i Y_i \qquad (2.12)$$

$$Max \qquad Z_2 = \sum_{i \in I} a_i r_i \qquad (2.13)$$

$$sujeito \quad a$$

$$r_i + Y_i \leq \sum_{i \in N_i} X_j \qquad \forall i \in I \qquad (2.14)$$

$$r_i \leq Y_i \qquad \forall i \in I \qquad (2.15)$$

$$\sum_{j \in J} X_j = p \qquad (2.16)$$

$$X_j, r_i, Y_i \in \{0,1\} \qquad \forall j \in J, i \in I$$

As funções objetivo maximizam a primeira cobertura e a segunda cobertura, respetivamente. A primeira restrição (2.14) afirma que a cobertura verificada pelo primeiro e segundo servidor está limitada pelo número de servidores que circundam o ponto de procura i. A segunda restrição(2.15) indica a cobertura de "*backup*" que não chega a ser prevista sem previamente se obter uma primeira cobertura.

Talvez, os modelos de localização estocástica mais populares sejam os que incorporam probabilidades. Por exemplo, Daskin ([16],[17]), numa das suas conhecidas extensões para o problema "*Maximal Covering*" assume que os servidores estão ocupados de acordo com uma dada probabilidade. O objetivo é maximizar a procura coberta pelos outros servidores que não se encontram ocupados.

Tendo em conta a formulação desse problema, considera-se q a probabilidade de um servidor estar ocupado. O aumento na cobertura esperada para determinada solicitação causada pela adição do k^{ésimo} servidor é dada por.

$$H_k = P_k - P_{k-1} = \left(1 - q^k\right) - \left(1 - q^{k-1}\right) = (1-q)q^{k-1} \qquad (2.17)$$

As variáveis são definidas como

$$Y_{ik} = \begin{cases} 1 & \text{se nó i tem pelo menos k servidores na vizinhança} \\ 0 & \text{caso contrário} \end{cases}$$

X_j número de servidores na localização j

O modelo maximiza a cobertura esperada para as restrições habitualmente utilizadas:

$$Max \quad Z = \sum_{i \in I} \sum_{k=1}^{n_i} a_i (1-q) q^{k-1} Y_{ik} \qquad (2.18)$$

sujeito a

$$\sum_{k=1}^{n_i} Y_{ik} \leq \sum_{j \in N_i} X_j \qquad \forall i \in I \quad N_i = \{ j | t_{ji} < S \} \qquad (2.19)$$

$$\sum_{j \in J} X_j = p \qquad (2.20)$$

$$Y_{ik} \in \{0,1\} \qquad \forall i \in I \quad \forall k$$

$$X_j \text{ inteiro} \qquad \forall j \in J$$

ReVelle e Hogan [7] formularam um modelo semelhante no qual maximizavam o número de nós através de múltipla cobertura.

Tan, Cheong e Goh [18] consideram o problema da afetação de veículos, com uma determinada capacidade de transporte, de um ponto central a um conjunto disperso de clientes com uma procura associada. Sendo conhecida à *priori* a capacidade dos veículos e a restrição temporal, apenas a procura é considerada estocástica. Estes autores apresentam um algoritmo multiobjectivo que incorpora uma heurística e simulação de rotas para avaliar a sustentabilidade das soluções.

Por sua vez, Özdemir, Yücesan e Herer [19] estudam a coordenação entre a localização de *stocks* através de estratégias de reabastecimento que tenham em conta a transferência de um produto entre localizações. É utilizada a simplificação de que o tempo de transferência é nulo, sendo deste modo desenvolvida uma solução baseada na perturbação infinitesimal que resolve o problema de otimização estocástico.

2.3. Modelos de Localização com Filas de Espera: servidores móveis.

A perspetiva que incorpora as interações ocorridas em filas de espera com modelos de localização não evidencia tantos desenvolvimentos; não obstante, alguns avanços interessantes têm vindo a ser formulados nas últimas décadas no que respeita a localização/afetação de serviços de emergência. O trabalho pioneiro em relação a este tópico foi o de Larson [20], que já considera um sistema de serviço estocástico espacialmente distribuído com servidores móveis igualmente dispersos – o modelo "*hypercube queuing*". Batta et al [21] recorreu a este modelo para demonstrar que a suposição implícita da independência do servidor, tal como assumida por Daskin [16],[17], é frequentemente violada.

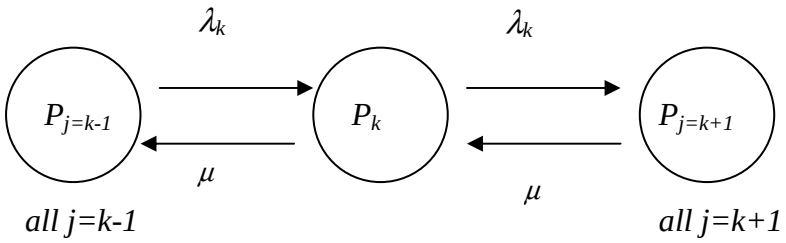
O modelo Hypercube

Um dos primeiros trabalhos que se preocupam com o congestionamento foi o modelo "*hypercube*" de Larson [20]. Este modelo foi classificado, de acordo com a notação de Marianov e ReVelle [22], como um modelo de filas de espera descritivo no sentido em que o utilizador inicia por especificar as localizações dos servidores para, de seguida, encontrar a solução com as distribuições em estado estacionário.

Os modelos "*hypercube*" de Larson consideram servidores móveis em sistemas congestionados. Cada servidor tem dois estados: ocupado (1) e livre (0). O sistema como um todo tem 2^p estados, dado existiram *p* servidores móveis a considerar. O posicionamento geográfico inicial dos *p* servidores é conhecido. A taxa de chamada (ou solicitação de serviço) em cada ponto de procura é descrita segundo um processo Poisson (λ_i) sendo os tempos de serviço Exponenciais

$\left(\frac{1}{\mu}\right)$. Cada servidor tem a sua área primária de serviço mas poderá atender solicitações que advenham de qualquer outro nó de procura. O sistema pode ser tratado como um sistema de filas de espera M/*M/p*.

Considere-se o estado b_s como "existem b_s servidores ocupados no sistema". Considere-se P_k como a probabilidade de estar no estado k e w_k a ponderação associada ao estado k. O diagrama de transações é ilustrado da seguinte forma:



E as "*Balance Equations*" são dadas por:

$$\sum_{j=k-1} P_j \lambda_k + \sum_{j=k+1} P_j \mu = P_k \lambda_k + P_k w_k \mu \qquad (2.21)$$

ou

$$P_k \left(\lambda_k + w_k \mu\right) = \sum_{j=k-1} P_j \lambda_k + \sum_{j=k+1} P_j \mu \qquad (2.22)$$

Uma das equações deve ser retirada e substituída por outra que force as probabilidades a serem iguais a um, sendo o processo iterativo resultante dado por:

$$P_k{}^n \left(\lambda_k + w_k \mu\right) = \sum_{j=k-1} P_j{}^{n-1} \lambda_k + \sum_{j=k+1} P_j{}^{n+1} \mu \qquad (2.23)$$

Larson [20] formulou uma aproximação ao modelo "*hypercube*", em que um sistema de apenas p equações necessita ser resolvido em vez das 2^p equações anteriormente enunciadas. Larson assume $\mu=1$; que cada solicitação é atendida por um e apenas um servidor e que os servidores atendem a solicitações originárias de qualquer ponto de procura, mas cada um destes possui uma área primária de serviço e uma lista ordenada por ordem de preferência dada a política de serviço e independente do estado vigente do sistema.

No trabalho de Hakimi [2], o problema 1- mediano localiza um servidor numa determinada rede de forma a minimizar a distância média percorrida entre o servidor e a população que representa a procura pelo serviço. Frequentemente, o tempo médio de espera em fila é muito maior que o tempo médio de deslocação e, posto isto, poderá ser relativamente mais importante tendo em conta a eficiência do serviço prestado. Adicionalmente, a magnitude do atraso em fila pode ser bastante sensível à localização da instalação servidora sendo esta a razão pela qual Berman et. al. [23], em parte motivado pelo estudo de Larson (modelo "*hypercube*"), aprofundou o problema proposto por Hakimi recorrendo ao procedimento geral no que concerne as filas de espera. De acordo com o problema, a formulação da procura por serviço será originada apenas nos nós de uma rede e ocorrerão no tempo de acordo com um processo de chegadas descrito por uma distribuição Poisson. Os autores consideram a localização de um único servidor que alberga um servidor móvel.

Batta e Mannur [24] formulam um modelo que tem em conta a realidade do serviço prestado por unidades móveis (carros de bombeiros ou ambulâncias). Estes autores examinam o problema da cobertura e problema da localização de máxima cobertura num contexto em que múltiplas unidades são condicionadas a diferentes pontos de procura.

Em 1993, Ball e Lin [25] propõem um modelo de localização de veículos de emergência com recurso à otimização de programação inteira binária (0-1), a qual mostrou ser uma técnica altamente eficaz.

Mandell [26] formula modelos de cobertura, igualmente para serviços de emergência médica, de forma a maximizar o número esperado de atendimento de chamadas de serviço tendo em conta a disponibilidade do servidor através de um modelo de filas de espera bidimensional.

Jamil, Baveja e Batta [27] concentraram-se no desenvolvimento de um modelo que visa um único centro de serviço e a respetiva localização, operando como uma fila M/G/1 e tendo como objetivo minimizar a combinação linear ponderada do quadrado do tempo de resposta médio e a variância do mesmo.

Por sua vez, os autores Berman e Vasudeva [28] consideram o problema de localizar um determinado número de centros de serviços (móveis) quando estes regressam à "base" dada a inexistência de chamadas, situação esta em que ficam a aguardar a respetiva distribuição de serviço.

Branas e Revelle [29] concentram-se no modelo "*Trauma Allocation Model for Ambulances and Hospitals*" e utilizam um algoritmo que combina um programa linear "*mixed-integer*" com uma nova heurística. Este modelo considera dois tipos de recursos e dois níveis hierárquicos tendo como objetivo a maximização da cobertura.

Por seu turno, Harewood [30] oferece uma versão multiobjectivo do problema "*Maximum Availability Location*" com uma aplicação real no contexto da afetação de ambulâncias na ilha de Barbados, Caraíbas. O primeiro objetivo do modelo é maximizar a população coberta dentro

de uma distância padrão e com um determinado nível de confiança. Em segundo lugar, o autor pretende que o modelo escolha as localizações que minimizam o custo de cobrir essa população.

2.4. Modelos de Localização com Filas de Espera: servidores fixos.

Servidores fixos podem também "sofrer" congestionamento. Este será o caso de serviços de saúde e, de uma forma geral, serviços públicos de qualquer natureza que tenham servidores fixos. Consultar como exemplo Marianov e Serra [31].

Berman, Larson e Chiu [32] efetuaram o trabalho considerado como o início do "casamento" entre as teorias de localização e teorias de filas de espera. Estes expandiram o trabalho realizado por Hakimi [2], no que respeita o problema 1-mediano, ao incorporá-lo no contexto de filas de espera. Na localização dos centros de serviço é explícita a sua dependência em relação aos tempos de serviço, tempos de deslocação e atrasos resultantes das próprias filas. Como base para o seu trabalho está presente, também, o modelo pioneiro desenvolvido por Larson [20] – *Hypercube Queuing Model*.

Batta, Larson e Odoni [33], no trabalho que desenvolveram, alertam para o facto do tratamento de filas de espera mais correntemente utilizado como ferramentas de decisão (nomeadamente FCFS – *first come, first serve*; LCFS – *last come, first serve*; SRO – *service in random order*) não ser o mais apropriado quando se considera a realidade do serviço com diferentes prioridades.

Por sua vez, Batta [34] considera o problema de localizar um único centro de serviço numa rede que opera como uma fila M/G/1, em que as chamadas em espera são atendidas de acordo com

uma classe de disciplinas de filas de espera que dependem unicamente da informação sobre o tempo esperado de serviço.

Brandeau e Chiu [35] desenvolveram o "*Stochastic Queue Center Location Model*" dedicado a sistemas congestionados que têm como objetivo minimizar o tempo máximo de resposta a qualquer consumidor, sendo que o tempo esperado de reposta tem em linha de conta o tempo de espera até o servidor estar livre mais o tempo de deslocação ao centro de atendimento.

O Problema de Cobertura Probabilística

ReVelle e Hogan's [7] apresentam a versão probabilística do problema de localização sendo esta a primeira abordagem ao congestionamento recorrendo a restrições probabilísticas explicitas na programação matemática. Os autores oferecem uma estimativa local para a fração de ocupação:

$$q_i = \frac{\bar{t} \sum_{k \in M_i} f_k}{24 \sum_{j \in N_i} X_j} = \frac{\rho_i}{\sum_{j \in N_i} X_j} \qquad (2.24)$$

onde,

$\bar{t}$ duração de uma chamada em horas
f_k frequencia de chamadas no nó k (chamadas por dia)
M_i conjunto de nós de procura localizados até S do nó i
$N_i = \{j | t_{ij} \leq S\}$
ρ_i rácio de utilização

Recorrendo a esta notação, a probabilidade de pelo menos um servidor estar livre dentro do tempo limite S será dada por:

$$1 - P(\text{todos os servidores do nó i ocupados}) = 1 - \left(\frac{\rho_i}{\sum_{j \in N_i} X_j} \right)^{\sum_{j \in N_i} X_j} \qquad (2.25)$$

Tendo em vista a formulação do problema de localização probabilística, os autores consideram um valor limite α para a probabilidade e incluem-no como restrição. Não existindo equivalente linear a esta restrição, ReVelle e Hogan apresentam a seguinte equivalência numérica determinística:

$$\sum_{j \in N_i} X_j \geq b_i \qquad (2.26)$$

em que b_i é o número inteiro mais pequeno que satisfaz:

$$1 - \left(\frac{\rho_i}{b_i} \right)^{b_i} \geq \alpha \qquad (2.27)$$

O problema de localização probabilística com estimativas locais para as frações de ocupação pode ser resumido da seguinte forma:

$$Min \quad Z = \sum_{j \in J} X_j \qquad (2.28)$$

$$sujeito \quad a$$

$$\sum_{j \in N_i} X_j \geq b_i \qquad N_i = \{ j | t_{ji} < S \} \qquad (2.29)$$

$$X_j = \text{inteiro não negativo}, \forall j \in J$$

Mais tarde, os mesmos autores [7] desenvolveram o problema "*Maximum Availability Location*" onde se maximiza a população com pelo menos b_i servidores. A formulação do modelo é a seguinte:

$$Max \quad Z = \sum_{i \in I} a_i Y_{ib_i} \tag{2.30}$$

sujeito a

$$\sum_{k=1}^{b_i} Y_{ik} \leq \sum_{j \in N_i} X_j \qquad \forall i \in I \quad N_i = \{j | t_{ji} < S\} \tag{2.31}$$

$$Y_{ik} \leq Y_{ik-1} \qquad \forall i \in I, \quad k = 2,3,...,b_i \tag{2.32}$$

$$\sum_{j \in J} X_j = p \tag{2.33}$$

$$Y_{ik}, X_j \in \{0,1\} \qquad \forall i \in I \quad \forall j \in J \quad \forall k$$

As variáveis são definidas como:

$$Y_{ik} = \begin{cases} 1 & \text{se k instalações são potenciais servidores do nó i} \\ 0 & \text{caso contrário} \end{cases}$$

$$X_j = \begin{cases} 1 & \text{se existir um servidor em j} \\ 0 & \text{caso contrário} \end{cases}$$

A restrição (2.31) afirma que existem no máximo tantos potenciais servidores como instalações de serviço na vizinhança do nó i. A restrição (2.32) força o nó a ter k-1 servidores antes de se verificar k servidores.

Ball e Lin [25] introduziram no modelo um limite superior para a probabilidade de não cobertura para cada ponto de procura, sendo este obrigado a verificar um valor inferior em relação ao pré-especificado.

Problema de Cobertura Probabilística com Filas de Espera

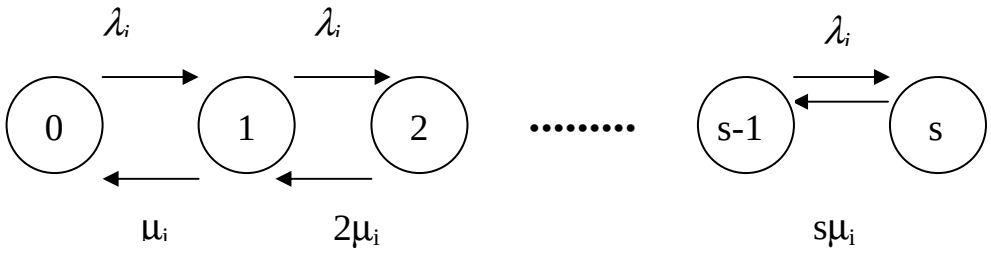
No trabalho desenvolvido por Marianov e ReVelle [22], “Problema de Cobertura Probabilística com Filas de Espera”, as restrições de segurança/confiança são formalmente introduzidas recorrendo à teoria de filas de espera para modelar o processo de partida-chegada. Como é evidenciado pelos autores, a distinção em termos de contribuição desta pesquisa envolve o uso

de uma estrutura probabilística aceite num modelo de otimização para um problema de localização.

O "Problema de Cobertura Probabilística com Filas de Espera" parte de duas importantes hipóteses: a densidade da procura varia unicamente de forma lenta de uma vizinhança para outras adjacentes; e o tempo de deslocação de uma vizinhança para outra é pequeno quando comparado com o tempo de serviço. Ao considerar estas premissas, os fluxos de servidores em ambas as direções cancelam-se mutuamente e a vizinhança fica como estando isolada, como uma unidade independente na qual todos os nós de procura e servidores interagem entre si sem interferirem com as procura ou servidores localizados fora desta vizinhança.

No que respeita as suposições relativas às chamadas de serviço e tempos de serviço, os autores modelam o comportamento de cada vizinhança como um sistema de filas de espera M/M/s/s (taxa de chegada de chamadas segundo uma Poisson, tempos de serviço distribuídos segundo uma Exponencial, *s* servidores, e até *s* chamadas sendo servidas simultaneamente). As chamadas geradas aquando da ocupação simultânea dos servidores são perdidas sob o ponto de vista da respetiva vizinhança.

Seja *s* o número de servidores na vizinhança. Definindo o estado *k* como *k* servidores ocupados, o diagrama de estado que resulta é o seguinte:



Está implícito neste diagrama que, considerando as hipóteses descritas a taxa de transição do estado *k* para o estado *k+1* é sempre igual independentemente do estado *k* no qual o sistema se

encontra (a taxa de chegada das chamadas de serviço não é afetada pelo número de chamadas que se encontram a ser servidas na altura).

A probabilidade *Pk* do sistema se encontrar no estado *k* é calculada recorrendo às convencionais equações estacionárias da teoria de filas de:

$$\left[P_{k-1}\lambda_i + (k+1)\mu_i P_{k+1}\right] - \left[P_k \lambda_i + k\mu_i P_k\right] = 0 \qquad (2.34)$$

para os estados 1,2,3,...,s; enquanto para o estado 0,

$$\mu_i P_1 - P_0 \lambda_i = 0 \qquad (2.35)$$

Ao resolver estas equações, resulta na seguinte estimativa para *Ps*, a probabilidade de todos os *s* servidores estarem ocupados:

$$P_s = \frac{\frac{1}{s!}\rho_i^s}{1 + \rho_i + \frac{1}{2!}\rho_i^2 + \ldots + \frac{1}{s!}\rho_i^s} \qquad (2.36)$$

Esta é a fórmula de Erlang para uma perda no sistema.

No seu modelo, Marianov e ReVelle requerem a probabilidade de pelo menos um servidor estar livre num determinado prazo que deverá ser superior a α, ou:

$$P_s = \frac{\frac{1}{s!}\rho_i^s}{1 + \rho_i + \frac{1}{2!}\rho_i^2 + \ldots + \frac{1}{s!}\rho_i^s} \leq 1 - \alpha \qquad (2.37)$$

Ps é função decrescente de *s*. Desta forma, *s* é o número de servidores que se requer estarem desocupados com uma probabilidade maior ou igual a *bi*, onde *bi* é menor número inteiro que satisfaz:

$$\frac{\frac{1}{s!}\rho_i^{b_i}}{1+\rho_i+\frac{1}{2!}\rho_i^2+...+\frac{1}{s!}\rho_i^{b_i}} \leq 1-\alpha \qquad (2.38)$$

De modo a facilitar o cálculo de *bi*, os autores oferecem uma fórmula recursiva que retribui o valor *Ps* quando o número de servidores é *s*, e como uma função de *Ps-1* quando o número de servidores é *s-1*:

$$P_s = \left(\frac{1}{P_{s-1} + \frac{s\mu_i}{\lambda_i}} \right) P_{s-1} \qquad (2.39)$$

A formulação é idêntica à seguida no "Problema de Cobertura Probabilística". Os autores calculam o número *s* de servidores que fazem a probabilidade ser menor que um dado valor, sendo que este número passa a ser o parâmetro *bi* do problema "Problema de Cobertura Probabilística com Filas de Espera".

O Modelo de Localização-Afetação de Cobertura Máxima com Filas de Espera

O "Modelo de Localização-Afetação de Cobertura Máxima com Filas de Espera" foi desenvolvido por Marianov e Serra [4]. Estes definem o objetivo do modelo como sendo o de "localizar *p* centros e afetar utilizadores a estes de forma a maximizar a população coberta, onde se define cobertura como: (i) população coberta é afeta a um centro dentro de um tempo ou distância específicas a partir da sua localização inicial, e (ii) caso um utilizador à sua chegada

seja coberto por um centro ocupado, esperarão em fila não mais do que *b* outros indivíduos com uma probabilidade de pelo menos α" ou alternativamente "(ii) caso um utilizador à sua chegada seja coberto por um centro ocupado, este será servido dentro do tempo τ desde a sua chegada, com uma probabilidade de pelo menos α".

O problema de Localização de Cobertura Máxima de Church e ReVelle [13] é então modificado a fim de acomodar as restrições de congestionamento:

$$Max \quad \sum_{i\in I}\sum_{j\in J} a_i X_{ij} \tag{2.40}$$

sujeito a

$$X_{ij} \le Y_j \qquad \forall i \in I, \forall j \in J \tag{2.41}$$

$$\sum_{j\in J} X_{ij} \le 1 \qquad \forall i \in I \tag{2.42}$$

$$P(\text{centro j tem} \le \text{b pessoas em fila}) \ge \alpha \quad \forall j \in J \tag{2.43}$$

$$\text{ou } P(\text{tempo espera no centro j} \le \tau) \ge \alpha \quad \forall j \in J \tag{2.44}$$

$$\sum_{j\in J} Y_j = p \tag{2.45}$$

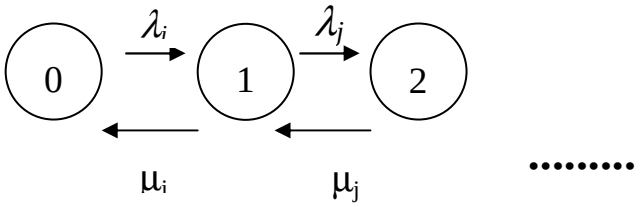
$$Y_j \in \{0,1\}, X_{ij} \in \{0,1\} \qquad j \in N_i \qquad \forall i, j$$

Os autores aceitam a suposição de que a procura de serviço em cada ponto de procura seja definida de acordo com um processo Poisson, com intensidade f_i. Se cada centro serve um conjunto de nós de procura, os pedidos de serviço nesse centro serão a reunião dos pedidos de serviço dos nós do conjunto. Podem assim ser descritos com um processo estocástico igual à soma de vários processos Poisson. Como o novo processo estocástico é também descrito segundo uma Poisson, tem uma intensidade λ_j igual à soma das intensidades do conjunto de nós servido pelo centro. Este conjunto de nós resultará da solução do problema. As variáveis X_{ij} são utilizadas para reescrever o parâmetro λ_j:

$$\lambda_j = \sum_{i\in l} f_i X_{ij} \tag{2.46}$$

Se uma determinada variável X_{ij} é unitária, significando que o nó *i* se encontra afeto ao centro em *j*, a intensidade f_i correspondente será incluída no cálculo de λ_j . Os autores também assumem um tempo de serviço exponencialmente distribuído, com uma taxa média de μ_j. Assumindo o estado estacionário, cada centro pode ser descrito com o sistema de filas de espera M/M/1.

O diagrama de transição de estado para o sistema de filas de espera M/M/1 é dado por:



As probabilidades associadas para o estado estacionário são

$$P_k = (1-\rho_j)\rho_j^k \qquad (2.47)$$

sendo $\rho_j = \lambda_j / \mu_j$ (2.48)

Se representarmos P_k como a probabilidade de nos encontrarmos no estado *k*, a restrição que requer que o número de elementos em fila seja menor que *b* com a probabilidade α, será escrita como:

$$P_0 + P_1 + ... + P_{b-1} \geq \alpha \qquad (2.49)$$

Ou, utilizando o resultado anterior:

$$(1-\rho_j) + (1-\rho_j)\rho_j + (1-\rho_j)\rho_j{}^2 + ... + (1-\rho_j)\rho_j{}^{b+1} \geq \alpha \qquad (2.50)$$

$$(1-\rho_j)\sum_{k=0}^{b+1}\rho_j^k \geq \alpha \qquad (2.51)$$

que é equivalente a:

$$\left(1-\rho_j\right)\frac{1-\rho_j^{b+2}}{1-\rho_j} \geq \alpha \tag{2.52}$$

ou após alguma algebra

$$\rho_j \leq \sqrt[b+2]{1-\alpha} \tag{2.53}$$

e substituindo $\rho_j = \lambda_j / \mu_j$ (2.54)

$$\lambda_j \leq \mu_j \sqrt[b+2]{1-\alpha} \tag{2.55}$$

ou, usando a definição *λj*:

$$\sum_{i \in l} f_i X_{ij} \leq \mu_j \sqrt[b+2]{1-\alpha} \tag{2.56}$$

Os autores também deduzem uma fórmula operacional para a restrição de congestionamento alternativa utilizando a função distribuição de probabilidade para o tempo de espera num sistema M/M/1, como se apresenta:

$$f_w\left(w_j\right) = \left(\mu_j - \lambda_j\right) e^{-\left(\mu_j - \lambda_j\right) w_j} \tag{2.57}$$

e a distribuição acumulada:

$$P\left(w_j \leq \tau\right) = F_w\left(\tau\right) = 1 - e^{-\left(\mu_j - \lambda_j\right)\tau} \tag{2.58}$$

Definindo a probabilidade como sendo maior ou igual a α:

$$1 - e^{-\left(\mu_j - \lambda_j\right)\tau} \geq \alpha \tag{2.59}$$

$$e^{-\left(\mu_j - \lambda_j\right)\tau} \geq 1 - \alpha \tag{2.60}$$

$$-\left(\mu_j - \lambda_j\right)\tau \geq ln\left(1-\alpha\right) \tag{2.61}$$

$$\lambda_j \leq \mu_j + \frac{1}{\tau} ln(1-\alpha) \tag{2.62}$$

A expressão que resulta para a restrição para o tempo de espera é dada por:

$$\sum_{i \in I} f_i X_{ij} \leq \mu_j + \frac{1}{\tau} ln(1-\alpha) \tag{2.63}$$

Ambas as expressões são equivalentes lineares às expressões originais.

O Problema de Localização de Cobertura para Filas de Espera com Prioridades

O Problema de Localização de Cobertura para Filas de Espera com Prioridades assume afetação estática dos consumidores aos centros de serviço. Esta é a suposição típica para a localização de servidores fixos sendo que a procura se desloca ao centro para obter o serviço pretendido (Berman e Krass [36]). Há duas formas alternativas de considerar estas afetações estáticas: a primeira versão do modelo assume um ambiente de escolha direta, onde uma autoridade central dita a afetação do cliente a um centro; e uma segunda versão onde se assume a escolha realizada pelo utilizador, sendo a afetação feita por cada individuo aplicando uma regra de decisão, no nosso caso sempre o centro mais perto.

A primeira versão do modelo define afetações separadas para as diferentes prioridades que poderão ou não coincidir, i.e., um ponto de procura pode ser afeto a um centro em *j* para uma das prioridades e ao centro $k \neq j$ para os restantes casos. Uma possível formulação para o modelo é a que se segue (uma versão p-mediano do Modelo de Cobertura Máxima):

Max

$$Z = \sum_k \sum_i \sum_j a_i X_{ij}^{[k]} \qquad (2.64)$$

sujeito a

$$X_{ij}^{[k]} \le Y_j \qquad \forall i \in I, \forall j \in N_i, \forall k \qquad (2.65)$$

$$\sum_{j \in N_i} X_{ij}^{[k]} \le 1 \qquad \forall i \in I, \forall k \qquad (2.66)$$

$$\sum_j Y_j = p \qquad (2.67)$$

$$W_j^{[k]} \le \tau^{[k]} \qquad \forall j, \forall k \qquad (2.68)$$

$$X_{ij}^{[k]}; Y_j \in \{0,1\} \qquad \forall i \in I, \forall j \in N_i \qquad (2.69)$$

onde,

$X_{ij}^{[k]}$ é um quando o ponto de procura *i* é afeto a um centro em *j* para as urgências de prioridade *k* e zero caso contrário;

Y_j é um caso um centro esteja localizado em *j* e zero caso contrário;

$W_j^{[k]}$ é o tempo médio de espera para a classe de prioridade *k* no centro *j*;

Os parâmetros $\tau^{[k]}$ representam os limites impostos aos tempos de espera para a prioridade *k*;

p é o número de centros a serem localizados;

a_i é a população no ponto de procura *i*;

I é o conjunto de todos os pontos de procura;

Ni é o conjunto de centros localizados a uma distância menor ou igual a *d* desde o ponto de procura *i*.

O objetivo maximiza a cobertura para todas as prioridades. As restrições (2.65) afirmam que se a população *i* é afeta a um centro em *j* para prioridade *k*, então existe um centro localizado em *j*; (2.66) força cada ponto de procura a ser afeto a não mais do que um centro; (2.67) define o número de centros a serem localizados; (2.68) força o tempo médio de espera a ser inferior a um determinado tempo especificado. Adicionalmente, *j* forçosamente deve pertencer ao conjunto N_i

(2.69); o mesmo será dizer que, de modo a um ponto de procura ser coberto deve existir um centro localizado dentro da distância limite *d*.

O modelo acima descrito pode ser convertido num de "escolha realizada pelo utilizador" ao assumir que estes escolherão sempre o centro mais próximo, e adicionando o seguinte conjunto de restrições que reforçam esta hipótese:

$$X_{ij}^{[k]} \geq Y_j - \sum_{l \in C_{ij}} Y_l \qquad C_{ij} = \left\{ l \middle| d_{il} < d_{ij} \right\} \quad \forall i \in I, \forall j \in N_i \tag{2.70}$$

A restrição (2.70) foi originalmente introduzida por Rojeski e ReVelle [37] no contexto de um problema de localização com restrição orçamental. Estabelece que se *j* representa um centro disponível, sem existir outro próximo nas mesmas condições, a procura *i* deve ser afeta a *j*. Se *j* estiver disponível mas algum outro centro próximo deste está também desocupado, esta relação de modo algum condiciona a afetação.

Para uma discussão mais detalhada sobre restrições de afetação de acordo com proximidade ver como exemplo Gerrard e Church (1996) [38].

$$W^{[k]} = W_0 + \sum_{i=1}^{k} \overline{S}^{[i]} \overline{L}^{[i]} + \sum_{q=1}^{k-1} \overline{S}^{[i]} \overline{M}^{[i]} \tag{2.71}$$

Num sistema de filas de espera com prioridades assume-se que um consumidor ao chegar pertence à classe de prioridade r (r = 1,2,...,R); quanto menor for o índice da prioridade maior será a prioridade da respetiva classe. Considerem-se prioridade "*nonpreemptive*", i.e. um cliente no processo não é passível de ser posto fora do serviço e voltar de novo à fila de espera sempre que um cliente com prioridade mais alta surge. Cliente da prioridade *k* chegam de acordo com uma distribuição Poisson de taxa $\lambda^{[k]}$ por unidade de tempo e cada consumidor deste grupo tem

um tempo de serviço (S) selecionado independentemente da distribuição B_k (S) com média $\overline{S}^{[k]}$. Considere-se adicionalmente um disciplina de filas HOL (*Head of the Line* – Primeiro da Fila) em cada nível de prioridade. O tempo médio de espera no sistema para a classe de prioridade *k* é assim dividido em três componentes:

$$W^{[k]} = \frac{W_0}{(1-\sigma_k)(1-\sigma_{k-1})} \qquad \text{se} \quad 1-\sigma_k > 0 \tag{2.72}$$

$$= +\infty \qquad \text{caso contrário}$$

onde, W_0 é o tempo que se espera que o serviço dure para o cliente que ocupa o servidor no momento em que um novo utilizador (da classe de prioridade *k*) chega à fila de espera; $\overline{L}^{[i]}$ é o número esperado de utilizadores da classe de prioridade *i* que já se encontram em espera na fila na altura em que um novo utilizador aparece e $\overline{M}^{[i]}$ é o número esperado de utilizadores da classe de prioridade *i* que estão para chegar enquanto o recém-chegado cliente aguarda serviço.

O tempo médio de espera para serviços de prioridade *k* é dado pela expressão:

$$\sigma_k = \sum_{i=1}^{k} \rho^{[i]} \qquad \text{com} \quad \sigma_0 \equiv 0 \tag{2.73}$$

Onde

$$\rho^{[i]} = \frac{\lambda^{[i]}}{\mu^{[i]}} = S^{[i]}\lambda^{[i]} \tag{2.74}$$

e

$\mu^{[i]}$ é a taxa de serviço para a classe de prioridade *i*.

Neste contexto, a interpretação de ρ é como habitual a fração de tempo que o servidor está ocupado (desde que ρ<1).

Como referido anteriormente, W_0 corresponde ao atraso médio para o consumidor em causa dada a existência de outro utilizador em atendimento, e pode ser calculado recorrendo à seguinte fórmula (consultar Kleinrock [39]):

$$W_0 = \sum_{i=1}^{R} \frac{\lambda^{[i]}\overline{\left(S^{[i]}\right)^2}}{2} \tag{2.75}$$

onde $\overline{\left(S^{[i]}\right)^2}$ representa o segundo momento na distribuição do tempo de serviço.

Tendo em conta o modelo desenvolvido, os fatores de utilização são definidos pelo produto entre o tempo médio de serviço e a taxa de chegadas definida como a soma de todas as taxas de chamada para todos os pontos de procura afetos ao centro em *j*:

$$\rho_j^{[k]} = \overline{S_j^{[k]}}\lambda_j^{[k]} = \overline{S_j^{[k]}}\sum_i f_i^{[k]} X_{ij}^{[k]} \tag{2.76}$$

Onde $X_{ij}^{[k]}$ é uma variável binária (0,1) que define a afetação da procura do ponto i a um centro em j para serviços de prioridade k. $f_i^{[k]}$ é a taxa de chamadas de serviço de prioridade k para o ponto de procura i. Como exemplo, um valor $f_i^{[k]} = 0.08$ indica que no ponto de procura i o número de chamadas por unidade de tempo para serviços da prioridade k, em média, é igual a 8% da população total.

As restrições sobre o tempo de espera podem ser reescritas da seguinte forma:

$$W^{[k]} = \frac{W_0}{(1-\sigma_k)(1-\sigma_{k-1})} \leq \tau^{[k]} \tag{2.77}$$

sendo estas não-lineares. Assim, o modelo não pode ser resolvido pelo método "*branch and bound*" e algumas aproximações como a relaxação Lagrangeana ou algoritmos heurísticos devem ser utilizados.

Uma importante área de aplicação deste tipo de modelos relaciona-se com a localização de serviços de emergência, dada a imprevisibilidade do número e tempo das chegadas de chamadas de serviço e o efeito na performance do sistema face ao congestionamento que resulta de algumas instalações servidoras receberem muitas chamadas durante um certo período de tempo.

Também importante, porém menos trabalhada, é a aplicação do modelo na localização de uma cadeia retalhista, ou outro tipo de serviço, em que o total da procura para uma determinada instalação pode ser adversamente afetada quando a taxa de serviço diminui dado o congestionamento verificado.

Um subgrupo de temas relacionados com localização tem sido referenciado como Problemas de Localização com Procura Estocástica e Congestionamento. Ver como exemplo Berman e Krass [36]. Estes concentram-se em duas fontes de incerteza: a procura gerada por cada cliente e o tempo em que esta ocorre; e uma possível perda de procura dada a impossibilidade da instalação em providenciar um serviço adequado dado o congestionamento.

Gendreau, Laporte e Semet [40] consideram um problema de relocalização para uma frota de ambulâncias propondo uma modelação dinâmica e, paralelamente, recorrem a uma heurística "*Tabu Search*". Legato e Mazza [41] apresentam um modelo de filas de espera para as atividades logísticas relacionadas com a chegada, atracamento e partida de veículos a um terminal de contentores. Foi construído um modelo de simulação de acontecimentos discretos

com a representação da política de prioridades, a afetação das gruas e o tempo de chegada entre os veículos.

Quanto a Verter e Lapierre [42], estes consideram o problema de localizar serviços de prevenção. Os autores assumem, neste trabalho, a distância como a maior determinante bem como o facto de as pessoas escolherem o estabelecimento mais próximo.

Goldberg [43] faz uma revisão do desenvolvimento na área da investigação operacional no que concerne a afetação e planeamento de serviços de emergência e bombeiros.

Todas as meta-heurísticas a que se recorre nesta tese foram adaptadas para o caso particular que se propõem a resolver sendo avaliadas, com o recurso a exemplos numéricos gerados aleatoriamente, no que concerne o seu comportamento em termos de qualidade da solução e tempo de computação.

2.5. Modelos de Localização com Simulação.

Em 1997, Zaki, Cheng e Parker [44] propõem um modelo de simulação para a gestão de serviços de emergência, mais especificamente, para veículos de polícia por diferentes zonas com padrões de procura não homogéneos. Estes optam por afetar os recursos existentes de forma a reduzir o tempo de resposta das unidades de emergência para um valor especificado sem que haja relocalização uma vez que estes já se encontram afetos e seria muito dispendioso e ineficaz uma nova distribuição das mesmas.

Também nesse ano, Jung, Blau, Pekny, Reklaitis e Eversdyk [45] estudam a otimização via simulação de uma cadeia logística de abastecimento com a particularidade de inserirem a procura (com a possibilidade de considerar tempos de entrega, alterações de encomendas, entre outros) como um fator variável.

Pichitlamken, Nelson e Hong [46] apresentam um procedimento de seleção sequencial dedicado especialmente a algoritmos de otimização via simulação, aplicados a casos de simulação dispendiosa com apenas uma medida de performance. A Seleção Sequencial com Memória (SSM) garante a escolha da melhor (ou quase-melhor) alternativa, na medida em que reutiliza a informação passada evitando, assim, reamostragem. Neste cenário, as probabilidades associadas são atribuídas pelo utilizador.

Tyni e Ylinen [47] introduzem o uso de algoritmos genéticos a um sistema de controlo de elevadores de carros recorrendo à otimização multiobjectivo para um ambiente de controlo dinâmico em constante alteração. Este trabalho baseia-se no "*Evolutionary Standardised-Objective Weighted Aggregation Method*" com um controlador que funciona como um decisor final interativo.

No tópico de gestão de inventários, Schwartz, Wang e Daniel [48] abordam a otimização via simulação, envolvendo perturbações estocásticas no que diz respeito à variabilidade do fornecimento e procura, erro de previsão, restrições de inventário, capacidade de transporte, entre outros aspetos.

Litvak, Rijsberge, Boucherie e Houdenhoven [49] propõem uma solução para encontrar o número de camas fruto da falta de capacidade de certas instalações responderem à procura na secção de cuidados intensivos. O método analítico é baseado num modelo "*overflow*" utilizado em sistemas de telecomunicações. Recorrendo ao "*Equivalent Random Method*" (ERM) como

melhor alternativa à simulação, é possível quantificar o número de pacientes rejeitados por falta de capacidade.

Yeh, e Lin [50] mostram como a qualidade do serviço de um departamento de emergência hospitalar pode melhorar, recorrendo à simulação e a um algoritmo genético, ajustando os horários do pessoal existente sem ser necessário recorrer a novas contratações.

Capítulo 3

Heurística

Este é o processo utilizado perante a necessidade de desenvolver métodos alternativos de pesquisa, simultaneamente menos dispendiosos em termos de tempo de computação e memória. Apesar de alternativos, estes devem ser capazes de identificar a solução ótima ou uma próxima desse resultado.

No que concerne a sua definição, uma heurística, tal como postulado por Reeves [51], é um método que procura boas (i.e. quase-ótimas) soluções com um tempo de computação razoável. Assume-se assim que as soluções encontradas por estes métodos heurísticos nem sempre são capazes de garantir o ótimo e, possivelmente, poderão apresentar soluções não possíveis.

Não obstante, os métodos heurísticos ganharam importância no seio dos modelos de localização dada a complexidade de formulações mais realistas que, perante as técnicas de otimização combinatória, se mostravam de difícil resolução.

Teitz e Bart [52] introduziram uma das primeiras heurísticas utilizadas para resolver modelos de localização. Esta é vista como uma aproximação através de “troca” ou “substituição” a partir do princípio que move as instalações das suas posições atuais para outras posições ainda não utlizadas mantendo esta nova posição sempre que o objetivo final melhore. Quando uma solução melhorada é obtida, o processo de procura é readaptado à nova solução. O procedimento termina assim que nenhuma outra solução melhor seja atingida.

Uma das heurísticas mais populares é a "*Greedy Randomized Adaptive Search Procedure*" ou GRASP de Feo e Resende [53]. Esta é uma meta-heurística com múltiplos pontos de partida para problemas combinatórios onde cada iteração consiste em duas fases: construção e procura local. A fase de construção constrói uma solução possível, cuja vizinhança é investigada até que um ótimo local é encontrado durante a fase de procura local.

O GRASP tem sido aplicado a um vasto leque de problemas de investigação operacional e otimização industrial. Estes incluem problemas de escalonamento, rotas, lógica, partição, localização e "*layout*", teoria de grafos, afetação, transformação, transportes, telecomunicações, desenho assistido, sistemas elétricos, e desenho VLSI (ver como exemplo Resende [54]). Igualmente Festa e Resende [54] apresentaram uma extensa bibliografia anotada de literatura sobre o GRASP.

Outra meta-heurística (que não foi utilizada neste trabalho) tem sido aplicada em larga escala ao resolver problemas de localização é a "*Tabu Search*" de Glover [55]. Esta impõe restrições a determinados movimentos usando uma função memória de curto prazo. O "critério de desejo" define quando estes movimentos não são mais tabu (normalmente após um determinado número de iterações). Adicionalmente, uma função memória de longo prazo é utilizada de modo a diversificar a pesquisa para outras áreas do espaço de soluções.

Como alternativas às meta-heurísticas, outras aproximações baseadas na relaxação Lagrangeana são frequentemente utilizadas para atingir a solução em modelos de localização. Estas técnicas substituem o problema original por uma versão mais simples eliminando uma ou mais restrições para depois as adicionar após terem sido multiplicadas por um multiplicador Lagrangeano associado à função objetivo. A maior virtude deste método é o de providenciar limites inferiores e superiores para o valor da função objetivo.

Apresentando a notação que será utilizada nesta secção:

- *j* índice de localizações possíveis;
- *i* índice de nós de procura;
- D_j lista de potenciais pontos de localização dos serviços ordenados de acordo com a população total;
- *S* solução
- $\overline{S}$ complementar da solução
- *C* conjunto de pontos candidatos
- *p* número de serviços a localizar
- *n* número de nós de procura
- *inc_j* taxa total de chamadas à potencial localização do serviço *j*
- D_{ij} lista de nós de procura dentro da distância limite desde a potencial localização do serviço *j*

O GRASP, já referido anteriormente, é um processo iterativo com solução fiável construída de forma independente a cada iteração. Uma iteração no GRASP consiste em duas fases: uma fase de construção e uma fase de procura local. A seguir descrevemos um pseudo-código para o algoritmo GRASP.

Procedure GRASP (Max_iterations, Seed)
For **k = 1** to **Max_iterations** do
S←Greedy_Randomized_Construction(Seed, γ);
S←Local_Search(Solution);
Update_Solution(Solution, Best_Solution)
enddo
end GRASP

Pseudo-Código 1: Pseudo-Código GRASP

O procedimento na fase de construção, que nos devolverá uma solução inicial em cada iteração, é denominada $Greedy\_Randomized\_Construction(Seed,\gamma)$, e é uma função da raiz no gerador de números aleatórios e do parâmetro *gamma* que define quais as soluções que serão incluídas na RCL - *Restricted Candidate List*, a lista que contém as melhores soluções.

O desenvolvimento do $Greedy\_Randomized\_Construction(Seed)$ é a seguir demonstrado:

procedure *Greedy Randomized Construction* (Seed,γ)
{sort candidate sites by decreasing order of population}
$D_j \leftarrow Sort\_Candidate\_Sites(population);$
{initialize solution set}
$S: = \{\ \};$
$\overline{S}: = C;$
{while solution is not a complete solution}
while $|S| \neq p$ **do**
{loop over all candidate sites not in the solution list}
For j=1 to $\left|\overline{S}\right|$ **do**
{initialize parameters}
{restrict demand points list to the standard covering distance to site j}
$D_{ij} \leftarrow \{i \in D, d_{ij} \leq d\}$
{sort demand points by increasing distance to site j}
$D_{ij} \leftarrow Sort\_Demand\_Points(distance);$

{loop over demand points in set Dij}
For i=1 to $\left|D_{ij}\right|$ **do**
{sum frequencies at each demand point if waiting time limit is not reached}
If ($W\_j < \tau^j$ and ρ_j<1) **do**
$inc\_j := inc\_j + f\_i$;
actualize w_j;
actualize ρ_j;
Endif
Enddo

{construct the restricted candidate list}
$c^{max} := max\{inc\_j\};$
$RCL \leftarrow \{j \in \overline{S}, inc\_j \geq \gamma c^{max}\};$
{select randomly one site from the RCL}
$j^* \leftarrow Random\_Select(RCL);$
$S: = S \cup \{j^*\};$
$\overline{S}: = \overline{S} \setminus \{j^*\};$
{take the demand points allocated to j* out of the demand points list}
For i=1 to $\left|D_{ij^*}\right|$ **do**
D := D\$\{i \in D_{ij^*}\}$;
Enddo
Enddo
end *Greedy Randomized Construction*

Pseudo-Código 2: Pseudo-Código da Fase de Construção

O algoritmo começa por escolher os nós candidatos de acordo com as respetivas procuras/populações**.** Consideramos no nosso exemplo que todos os nós de procura são também potenciais localizações de serviços. Outra possibilidade consideraria apenas um subconjunto dos nós de procura da lista $D_{j.}$.

Sendo assim, começando com o primeiro nó candidato da lista, afetámos-lhe os nós de procura mais próximos até o limite de cobertura ser atingido. No nosso exemplo, o limite de cobertura pode ser atingido quer pelo coeficiente de utilização, quer pelo limite imposto para o tempo de espera.

A procura total afeta a cada uma das potenciais localizações *j* é denominada de *incoming call rate*. A *incoming call rate* funciona como a função "*greedy*" (tradução livre: gananciosa) do algoritmo e pode ser definida por uma ponderação dos nós de procura ainda não cobertos que o passarão a ser caso a localização *j* para o serviço fosse escolhida.

Incluímos na RCL - Restricted Candidate List (uma lista restrita, i.e., o subconjunto, que contém as melhores soluções) os nós candidatos com uma *incoming call rate* total maior ou igual a *gamma* por cento da *incoming call rate* correspondente à potencial localização com o maior valor.

No algoritmo GRASP, o parâmetro *gamma* é definido à priori. (por exemplo, caso o valor de *gamma* seja igual a 0.8, queremos dizer que incluímos na lista que contém as melhores soluções - *Restricted Candidate List* - todas as potenciais localizações com um total de chegada de chamadas de serviço maior do que 80% da maior taxa de chegada de chamadas).

Tenha em atenção que na heurística "*greedy*", como a sugerida por Marianov e Serra [4] , a escolha seria sempre localizar um centro no ponto com o maior somatório de taxa de chegada de chamadas, i.e., γ=1.

3.1. Solução Heurística.

A cada iteração, escolheríamos aleatoriamente, de entre as localizações candidatas com a maior taxa de chegada de chamadas (i.e., as incluídas na *Restricted Candidate List*) as p localizações para os serviços.

procedure *Local_Search* (Solution, Best_Solution)
obj_best : = *obj(S)*;
{loop over sites in the solution}

for all $j_1 \in S$ **do**
$S := S \setminus \{j_1\}$;
{loop over sites not in the solution}
for all $j_2 \in \overline{S}$ **do**
evaluate $obj(S \cup \{j_2\})$;
if *obj_best*< $obj(S \cup \{j_2\})$ **do**
$S := S \cup \{j_2\}$:
obj_best : = $obj(S \cup \{j_2\})$;
else
$S := S \cup \{j_1\}$:
endif
enddo
Enddo
end *Local_Search*

Pseudo-Código 3: Pseudo-Código da Fase da Procura Local

Na fase da procura local *("local search phase*"), para cada centro de cada vez, desafetámos a procura que lhe foi atribuída, e movemo-la para todas as localizações possíveis ainda não utilizadas, repetindo de cada vez os passos 9 a 20 do procedimento *Greedy Randomized Construction* tendo em vista avaliar o objetivo. Se algumas localizações

retribuírem um melhor objetivo, mantemos o centro de serviço nesse ponto; caso contrário, mantemo-la na localização inicial (consultar Pseudo-Código 3). Repetimos o procedimento até que não seja possível melhorar a solução inicial ou o limite de iterações seja atingido.

Num ambiente de escolha definida pelo utilizador, o algoritmo necessita ser adaptado, tanto na sua construção como na fase da procura local, de modo a reforçar uma afetação próxima. Nesta nova versão das heurísticas, um ponto de procura será sempre afetado ao centro mais próximo, facto este que poderá conduzir a soluções pouco fiáveis dado os limites impostos para o tempo médio de espera não serem verificados. O algoritmo proposto penaliza o objetivo final sempre que uma solução pouco fiável é obtida. No caso de se obter uma solução fiável, considera-se este conjunto de localizações como potenciais para a colocação dos centros de serviço. Caso contrário, considera-se este conjunto de localizações apenas como uma solução inicial e não como uma potencial localização de serviço penalizadora do objetivo com um grande valor negativo M. Isto irá corresponder ao seguinte procedimento de avaliação do objetivo:

```
procedure evaluate_objective (S)
    Allocate each demand point to its closest center location;
    Evaluate W_j and ρ_j;
    obj(S):=0;
    If (W_j <τ and ρ_j<1) do
                    For j=1 to p do
                        For i=1 to n do
                            If (i is allocated to j) do
                                obj(S):=obj(S) + f_i;
                            endif;
                        enddo;
                    enddo;
    Else
            obj(S):=M;
    end evaluate_objective;
```

Pseudo-Código 4: Pseudo-Código da Avaliação do Objetivo

Na Fase da Procura Local (*Local Search Phase*), para cada centro de cada vez, desafetam-se as procuras que lhe foram atribuídas e passamo-las para todas as potenciais localizações ainda não

utilizadas. Alocámos sempre um nó de procura à localização potencial mais próxima e verificamos a possibilidade no que concerne o limite para o tempo de espera. Caso a solução encontrada não seja possível, penalizamos o objetivo com um valor negativo muito alto M. Sempre que novas afetações resultem num objetivo melhor, mantemos o centro nessa respetiva localização. Caso contrário, mantemos a localização inicial. Este procedimento é repetido até que não haja uma solução melhor que a anterior.

Os métodos heurísticos apresentados serão aplicados a cada um dos problemas que nos propormos desenvolver nos capítulos seguintes.

Capítulo 4

Problema de Localização com Cobertura Máxima para Filas de Espera

A formulação do modelo apresentado neste trabalho segue de perto a metodologia apresentada por Marianov e Revelle [22]. Os autores relaxam a condição de dependência entre a disponibilidade do servidor e o comportamento do modelo em cada uma das regiões como um sistema de filas M/M/s-*loss*. Desta forma é possível obter uma formulação probabilística do conjunto de localizações para o problema de cobertura. As restrições de segurança são introduzidas utilizando teoria de filas de espera para modelar o processo de chegadas e partidas dentro do próprio modelo de localização.

Os autores referem-se a centros de serviço fixos ao contrário das pesquisas iniciais que consideram os servidores móveis. Na mesma linha de análise, Marianov e Serra [4] apresentam vários modelos – probabilísticos, cobertura máxima, localização/afetação – com tempo de espera condicionado ao comprimento da fila.

Nesta secção, propomos um modelo que faz ligação entre o modelo "Localização-Afetação de Cobertura Máxima" e a Teoria de Filas de Espera. Na secção 4.1. descrevem-se os resultados das filas de espera que serão incluídos no modelo. Em 4.2. explicámos este novo modelo. A secção 4.3. formula o Problema da Localização com Cobertura para Filas de Espera particularizando para os casos em que a escolha é feita por um decisor central – Escolha Direta 4.3.1. – e os casos em que a escolha do centro prestador de serviço é realizada pelo próprio consumidor – Escolha do Cliente 4.3.2. Em 4.4. descreve-se o algoritmo proposto para

posteriormente, em 4.5. e 4.6. respetivamente, após ser levada a cabo a Experiência Computacional serem tiradas algumas Conclusões.

4.1. Resultados da Teoria de Filas de Espera.

As Filas de Espera são estudadas num grande número de manuais e trabalhos de investigação. Neste trabalho em concreto seguimos a notação utilizada no manual de Kleinrock [39].

Os consumidores/clientes chegam de acordo uma distribuição de Poisson à taxa λ por unidade de tempo e cada consumidor pertencente a este grupo vê o seu tempo de serviço (S) selecionado independentemente da distribuição B(S) com média S.

Consideremos um procedimento HOL (Head of the Line – Primeiro da Fila). O tempo médio de espera para o serviço é dado pela seguinte expressão:

$$W = \frac{\rho}{\mu(1-\rho)} \qquad (4.1)$$

e

$$\rho = \frac{\lambda}{\mu} = S\lambda \qquad (4.2)$$

sendo μ a taxa de serviço.

A interpretação do parâmetro ρ é, como normalmente, a fração de tempo em que o servidor está ocupado (enquanto ρ <1).

4.2. Definição do Problema.

A estrutura do nosso problema inclui um espaço discreto em que os consumidores (representando a procura) são posicionados e onde os centros de serviço estão por ser localizados por um agente decisor.

É assumido que os consumidores estão posicionados nos nós da rede (nós de procura). O processo de chamadas de serviço para cada nó *i* decorre de acordo com um processo Poisson com uma taxa λ_i.

O problema localizará um dado número de centros que providenciarão o atendimento. Assumimos, também, que é identificado o conjunto elegível (discreto) de possíveis localizações dos centros.

Tendo em conta os modelos a desenvolver, o fator de utilização é definido pelo produto entre o tempo médio de serviço e a taxa de chegada, sendo esta definida pelo somatório das taxas de chamadas de todos os pontos de procura afetos a um centro de serviço em *j*:

$$\rho_j = \overline{S_j}\lambda_j = \overline{S_j}\sum_i f_i \; X_{ij} \qquad (4.3)$$

X_{ij} é uma variável binária (0-1) que define a afetação da procura no ponto *i* ao centro de serviço em *j*, sendo que f_i representa a taxa de chamadas de procura de serviços no ponto de procura *i*.

Por exemplo, o valor de $f_i = 0.1$ indica que, no ponto de procura *i*, o número de chamadas por unidade de tempo representa 10% da população total, em média.

4.3. Formulação do Problema.

Os modelos de cobertura são frequentemente aplicados ao caso dos serviços públicos. Os objetivos neste tipo de serviços, no que respeita a localização de instalações servidoras, têm principalmente a ver com a minimização de custos sociais, universalidade do serviço (i.e., cobertura máxima), eficiência e equidade. No nosso caso, o objetivo é o de, na solução de localização encontrada, tentar abranger o máximo de população possível dado o número de instalações servidoras possíveis de implementar.

A formalização do problema proposto é a seguinte:

$$Max \quad Z = \sum_{i \in I} a_i Y_i \tag{4.4}$$

sujeito a

$$Y_i \leq \sum_{j \in N_i} X_j \qquad \forall i \in I \quad N_i = \{j | t_{ji} < S\} \tag{4.5}$$

$$\sum_{j \in J} X_j = p \tag{4.6}$$

$$X_j \in \{0,1\} \qquad \forall i \in I, \forall j \in J$$

em que,

a_i procura no nó i

$$Y_i = \begin{cases} 1, \text{caso o nó de procura esteja coberto} \\ 0, \text{caso contrário} \end{cases}$$

$$X_j = \begin{cases} 1, \text{se um servidor está localizado em j} \\ 0, \text{caso contrário} \end{cases}$$

A primeira restrição (4.5) define a cobertura: o nó de procura *i* está coberto caso uma instalação esteja localizada dentro da distância limite *S*. A segunda restrição (4.6) fixa o número de instalações a serem localizadas.

O Problema de Localização com Cobertura para Filas de Espera assume a afetação estática de clientes a centros de serviço. Esta é uma suposição típica para o caso de localizações com servidores fixos em que os consumidores deslocam-se ao centro para obter o tratamento [36].

Há duas versões deste problema. Uma primeira assume um Ambiente de Escolha Direta, em que um decisor central fixa a afetação de um cliente a um centro. A segunda assume um Ambiente de Escolha do Cliente sendo que, neste caso, a afetação é feita de acordo com a escolha pessoal do cliente. Assumir-se-á que nesse caso este deslocar-se-á ao centro mais próximo.

4.3.1. Ambiente de Escolha Direta.

Esta versão do Problema de Localização com Filas de Espera definirá localizações para os serviços de modo a maximizar a população coberta.

São impostas restrições temporais sendo a formulação do modelo a que se segue:

$$^{Max}Z = \sum_i \sum_j a_i X_{ij}$$

sujeito a

$$X_{ij} \leq Y_j \qquad \forall \text{ i} \in I, \forall \text{ j} \in N_i \qquad (4.7)$$

$$\sum_{j \in N_i} X_{ij} \leq 1 \qquad \forall \text{ i} \in I \qquad (4.8)$$

$$\sum_j Y_j = p \qquad (4.9)$$

$$W_j \leq \tau \qquad \forall \text{ j} \qquad (4.10)$$

$$X_{ij};\ Y_j \in \{0,1\} \qquad \forall \text{ i} \in I, \forall \text{ j} \in N_i$$

$$N_i = \{j | d_{ij} \leq ldist\} \qquad (4.11)$$

$$W_j = \frac{\rho_j}{\mu(1-\rho_j)} \qquad (4.12)$$

onde,

- X_{ij} é 1 caso o ponto de procura *i* seja afeto a um centro e 0 caso contrário;
- Y_j é 1 se um centro é localizado em *j* e 0 caso contrário;
- W_j é o tempo médio de espera no centro *j* (ver equação 1), sendo $\rho_j = \frac{\lambda_j}{\mu}$ e $\lambda_j = \sum_i a_i X_{ij}$;
- O parâmetro τ representa os limites impostos de tempo de espera;
- *p* é o número de centros a serem localizados;
- a_i representa a população no ponto de procura *i*;
- *I* é o conjunto de todos os pontos de procura;
- N_i representa o conjunto de centros localizados a uma distância menor ou igual a *ldist* a partir do ponto de procura *i*;

A restrição (4.7) afirma que caso a população *i* seja afeta a um centro *j*, então existe um centro localizado em *j*; (4.8) obriga a que cada ponto de procura não seja afeto a mais do que um centro; (4.9) define o número de centros a serem localizados; (4.10) força a que o tempo médio de espera seja inferior a um limite pré estabelecido. Adicionalmente, *j* tem de forçosamente pertencer ao conjunto N_i. O mesmo será dizer que para que um ponto de procura seja coberto deverá existir um centro localizado dentro de um limite de distância *d*.

4.3.2. Ambiente de Escolha do Cliente.

O modelo descrito anteriormente (seção 4.3.1.) pode ser convertido de forma a prever a escolha do centro feita pelo próprio utilizador, assumindo que os consumidores/clientes escolherão

sempre o centro de serviço mais próximo. Reforçando esta suposição, acrescenta-se o seguinte grupo de restrições:

$$X_{ij} \geq Y_j - \sum_{l \in C_{ij}} Y_l \qquad C_{ij} = \left\{ l \middle| d_{il} < d_{ij} \right\} \quad \forall i \in I, \forall j \in N_i \tag{4.13}$$

A restrição (4.13) foi originalmente introduzida por Rojeski e ReVelle [37] no contexto do problema do orçamento médio restringido. Esta estabelece que caso j seja um centro disponível e que não existe nenhum outro aberto e mais perto, então a procura i deverá ser "desviada" para j. Se, por acaso, j estiver aberto mas houver também outro centro mais próximo que também esteja aberto esta relação não restringe a afetação de nenhuma forma.

Para uma discussão mais detalhada sobre restrições de afetação mais próxima ver como exemplo Gerrard e Church [38].

4.4. Descrição do Algoritmo.

Nesta seção pretende-se descrever o processo sequencial e iterativo, assumido em termos de computação, e que tem como objetivo a resolução do problema de localização em estudo com recurso a um processo heurístico. A descrição que se segue enuncia todos os passos e procedimentos recursivos impostos através de um programa de computador cujo algoritmo, inspirado no p-minmax de Daskin [9], foi programado em C++ para atingir uma solução para o problema que se espera quase ótima.

Assim sendo, inicia-se o sistema com a leitura do ficheiro distância reconhecendo-se nesta fase do processo a rede de pontos de procura - todos eles potenciais localizações - e a distância entre estes - nosso custo associado medido em termos de tempo.

Para cada nó representativo de um centro populacional, gera-se a respetiva população sendo esta obtida de acordo com uma distribuição Uniforme. Sendo este parâmetro definido pelo utilizador, tendo em vista sujeitar o modelo a diferentes condições, a procura é assim estimada com base em uma percentagem da população anteriormente obtida. Estes valores - população e procura - são gerados para cada um dos cenários (ns) utilizados e serão tantos quanto o número de cenários com os quais se pretende trabalhar.

Sendo que o sistema foi inicializado, ou seja, temos em nossa posse uma caraterização da rede no que respeita o número de nós, distância entre estes e respetivas populações e procuras, recorrendo ao *software* de otimização CPLEX, o modelo começa por resolver o problema de localização com cobertura máxima para cada cenário.

O valor ótimo para a nossa função objetivo é então obtido passando a funcionar como futura referência aquando da comparação com os resultados da heurística. Esta solução inclui: quais os nós onde se localizam os centros de serviço, qual a afetação dos pontos de procura aos respetivos centros e qual valor da solução objetivo no que concerne o total de população coberta/prevista.

Com base nesse padrão da solução ótima constrói-se uma matriz (ns, ns), que se intitula Regret 1, na qual cada linha e coluna representa um dos cenários simulados. Neste sentido, a diagonal dessas matrizes regret representa os valores ótimos obtidos em cada um dos cenários simulados. Os valores fora dessa diagonal, i.e., os valores adjacentes à solução ótima, são obtidos fazendo o cálculo da função objetivo considerando as caraterísticas dos outros cenários (populações e

procuras) mas mantendo o padrão de localizações e afetações retribuídos pelo *software* de otimização CPLEX.

197182	201432	198357	196556	195265	194325	198562	196663	197701	194003
161050	190007	181250	182356	192123	179125	183459	168971	179157	189457
170892	167845	157853	164887	169741	190187	178112	156746	192454	189451
171313	175949	194848	164073	169787	199454	185464	177989	184188	188774
169787	191717	188772	169745	180869	183556	197010	191141	186311	188745
196787	192776	193741	183797	181579	171896	181235	182454	192478	179878
197121	195798	177131	193457	189743	172656	164681	165888	178432	182747
177336	181656	189743	187141	191778	194331	184556	175624	185655	189774
179487	173998	167131	189477	183454	199466	193473	195471	164635	167979
189741	185731	159887	169741	200874	190157	182486	189478	168635	182916

Figura 1. Matriz Regret 1 para 10 cenários.

Uma chamada de atenção uma vez que o cálculo destes valores adjacentes à "diagonal de ótimos" necessitam de um teste de possibilidade para que, na continuação do algoritmo, se venha a obter obrigatoriamente uma solução inicial exequível. O teste de possibilidade de uma solução adjacente à ótima deverá levar em linha de conta:

i. Que o limite de distância (ldist) do ponto de procura a afetar ao centro de serviço é respeitado;
ii. Que o limite de tempo de espera (wlim) é respeitado; e
iii. Alertar, omitindo-os do restante processo, quando tempos de espera negativos estiverem associados.

Nos casos em que o valor objetivo calculado, com o padrão de localizações e afetações, para os restantes cenários não for possível de acordo com as condições de possibilidade acima descritas, o valor que constará da matriz regret será zero sendo desta forma omitido da nossa heurística.

Findo o teste de possibilidade, com base na matriz Regret 1, constroem-se as matrizes Regret 2 e Regret 3, sendo cada uma destas uma transformação das anteriores, como a seguir se descreve:

Regret 2: cada valor desta matriz será obtido pela diferença entre o objetivo de um determinado cenário e o seu respetivo ótimo (retribuído pelo CPLEX e que consta da diagonal da matriz Regret 1). Desta forma, imperativamente na diagonal da matriz Regret 2 constarão apenas zeros. Este procedimento permite-nos assim obter os valores de um "Regret Absoluto".

0	4250	1175	-626	-1917	-2857	1380	-519	519	-3179
-28957	0	-8757	-7651	2116	-10882	-6548	-21036	-10850	-550
13039	9992	0	7034	11888	32334	20259	-1107	34601	31598
7240	11876	30775	0	5714	35381	21391	13916	20115	24701
-11082	10848	7903	-11124	0	2687	16141	10272	5442	7876
24891	20880	21845	11901	9683	0	9339	10558	20582	7982
32440	31117	12450	28776	25062	7975	0	1207	13751	18066
1712	6032	14119	11517	16154	18707	8932	0	10031	14150
14852	9363	2496	24842	18819	34831	28838	30836	0	3344
6825	2815	-23029	-13175	17958	7241	-430	6562	-14281	0

Figura 2. Matriz Regret 2 para 10 cenários.

Regret 3: a diferença obtida de acordo com os cálculos efetuados na matriz Regret 2 é agora dividida pelo valor ótimo de referência para o cenário em causa. Ficámos assim a conhecer o valor do "Regret Relativo" associado a cada cenário.

0,00000	0,02155	0,00596	0,00317	0,00972	0,01449	0,00700	0,00263	0,00263	0,01612
0,15240	0,00000	0,04609	0,04027	0,01114	0,05727	0,03446	0,11071	0,05710	0,00289
0,08260	0,06330	0,00000	0,04456	0,07531	0,20484	0,12834	0,00701	0,21920	0,20017
0,04413	0,07238	0,18757	0,00000	0,03483	0,21564	0,13037	0,08482	0,12260	0,15055
0,06127	0,05998	0,04369	0,06150	0,00000	0,01486	0,08924	0,05679	0,03009	0,04355
0,14480	0,12147	0,12708	0,06923	0,05633	0,00000	0,05433	0,06142	0,11974	0,04644
0,19699	0,18895	0,07560	0,17474	0,15219	0,04843	0,00000	0,00733	0,08350	0,10970
0,00975	0,03435	0,08039	0,06558	0,09198	0,10652	0,05086	0,00000	0,05712	0,08057
0,09021	0,05687	0,01516	0,15089	0,11431	0,21156	0,17516	0,18730	0,00000	0,02031
0,03731	0,01539	0,12590	0,07203	0,09818	0,03959	0,00235	0,03587	0,07807	0,00000

Figura 3. Matriz Regret 3 para 10 cenários.

É precisamente com base na matriz Regret 3 que prosseguimos o nosso algoritmo com a heurística sugerida por Daskin et al. [9]. Perante esta última matriz, os procedimentos serão os seguintes:

i. Da matriz Regret 3, observando os valores em linha, escolhe-se o que apresentar o regret relativo maior, ou seja, o que mais se afasta do ótimo de referência na diagonal obtido com recurso ao CPLEX;

ii. Posteriormente, de todos esses valores máximos escolhe-se o menor com o intuito de o utilizar com solução inicial na *local search* que se seguirá. Esta será alterada, no que respeita as localizações e afetações, tendo em vista a melhoria do objetivo até então apresentado de acordo com o já conhecido GRASP

O processo desenvolvido neste GRASP, após escolhida a solução inicial como se descreveu nos parágrafos anteriores, segue-se da seguinte forma:

1. Das localizações que constam da solução inicial obtida anteriormente, aleatoriamente, é escolhida uma que será retirada a fim de ser substituída por uma outra que, obrigatoriamente, deverá constar da RCL – *Restricted Candidate List*;
2. As potenciais localizações pertencentes à RCL devem preencher os requisitos de aceitabilidade no que respeita os limites de distância impostos;
3. Caso faça parte da RCL é aceite temporariamente para ser considerada no processo iterativo e, ao substituir a localização anterior, alterna a sua posição no que respeita a sua afetação aos pontos de procura;
4. Quando a solução inicial é melhorada, este novo padrão de localizações e afetações é aceite;
5. Caso contrário, mantém-se a solução inicial.

Em seguida, são novamente construídas as matrizes Regret 1, Regret 2 e Regret 3 a partir dos valores agora obtidos na *local search* realizada, tendo estas que igualmente passar pelos testes de possibilidade que oportunamente se descreveram. Este processo é repetido para um número pré-definido de iterações.

4.5. Experiência Computacional.

Sendo o objetivo observar a diferença entre os resultados, em termos de solução heurística e de solução inicial, foi realizada uma experiência relativamente simples. Esta consiste em gerar situações problemáticas aleatoriamente de forma a comparar os resultados da solução heurística e os resultados iniciais obtidos que funcionarão como ponto de partida (e comparação) para a *Local Search* do GRASP.

O problema de Localização com Cobertura Máxima explorado neste trabalho, tal como a heurística avaliada, baseiam-se no estudo de uma rede de pontos que representam centros de procura. A dimensão desta rede será variável e a cada centro será atribuída uma determinada frequência de procura (necessidade de serviço/atendimento).

As caraterísticas desta rede podem ser alteradas no que respeita, principalmente, o número de nós ou centros de procura.

Neste contexto, no trabalho levado a cabo, são geradas redes com 25, 40, 50 e 55 nós e a cada um destes é atribuída a respetiva frequência de procura. Para cada nó, é gerada, de acordo com uma distribuição Uniforme [800;1800], a população a utilizar como dado no sentido de obter a frequência de procura. Dada a população, 1% é considerada a frequência de procura (ou necessidade de serviço/atendimento).

Note-se que, em cada cenário gerado e para rede específica, a distância entre nós será constante dado que se altera apenas o tamanho da rede e a procura verificada em cada ponto da mesma. A distância entre pontos de procura é obtida com recurso a uma matriz distâncias comum a todos os cenários e redes estudadas.

Tendo como objetivo testar a heurística desenvolvida, além das diferenças de dados no que concerne a dimensão da rede, foram também fixados valores alternativos tanto para o Limite do Tempo de Espera como para o Limite de Distância, como se poderá verificar na Tabela 4.1.
No que respeita o processo recursivo do algoritmo, tendo em vista o apuramento da qualidade das conclusões, foram testados programas ora com um número diferente de iterações, ora com um número diferente de cenários, tal como se apresenta na próxima tabela.

Um resumo das caraterísticas e parâmetros dos conjuntos de dados trabalhados são a seguir apresentados na Tabela 4.1.

	Número		**Limites**		**Número**	
Casos	**Nós**	**Centros**	**Distância**	**Tempo Espera**	**Cenários**	**Iterações**
1	55	5, 10 e 20	10	0.02	10, 100 e 1000	100
2	55	5, 10 e 20	10	2	10, 100 e 1000	100
3	55	5, 10 e 20	10	20	10, 100 e 1000	100
4	55	5, 10 e 20	1	20	10, 100 e 1000	100
5	55	5, 10 e 20	10	20	10, 100 e 1000	100
6	55	5, 10 e 20	20	20	10, 100 e 1000	100
7	25	5, 10 e 20	10	0.02	10	500, 1000 e 2000
8	40	5, 10 e 20	10	0.02	10	500, 1000 e 2000
9	50	5, 10 e 20	10	0.02	10	500, 1000 e 2000

Tabela 4.1. Caraterísticas e parâmetros dos conjuntos de dados processados

O algoritmo em estudo foi implementado num computador com processador Dual 2.50 GHz Pentium Dual-Core com 1920 MB de memória e recorrendo ao compilador C++ Microsoft Visual Studio 2005 onde se integra, para a resolução dos problemas propostos, o *software* de otimização CPLEX *Optimization Studio* 12.2.

Em termos gerais, o nosso objetivo ao analisar os resultados obtidos é verificar até que ponto o padrão de localizações e afetações, retribuídos pela heurística, mostra haver ou não diferenças em relação ao padrão ótimo de localizações e afetações obtido na resolução do problema.

4.5.1. Alterando Limites para Tempo de Espera e Distância.

Na tentativa de interpretar o comportamento da heurística, foi utilizada uma rede com 55 pontos de procura com um total de 100 iterações. Varia, neste caso, o número de centros de serviço a localizar bem como o número de cenários estudados. Adicionalmente, com o intuito de gerar situações díspares para o sistema, tanto o limite de tempo de espera no servidor (wlim) como o limite de distância entre o servidor e o centro de procura (ldist) são alterados.

Os valores testados para o limite de tempo de espera no servidor foram wlim = 0.02, wlim = 2 e wlim = 20. Em todos estes casos o limite de distância entre o servidor e o centro de procura foi fixado em ldist = 10.

Para wlim = 0.02, independentemente do número de cenários e do número de centros fixados, a heurística retribuí sempre uma solução final idêntica à solução de partida (solução inicial) coincidindo assim as localizações iniciais e finais.

Já para o caso em que wlim = 2, obteve-se o maior número de casos em que as localizações finais diferiam das localizações iniciais. Acontece o descrito em 3 dos 9 casos testados, nomeadamente ao considerar a localização de 10 centros de serviço e simulando 100 cenários tal como ao tentar localizar 20 centros de serviço, para qualquer uma das simulações de 10 e 100 cenários.

Ao utilizar o limite de tempo de espera no servidor wlim = 20, apenas em 2 dos 9 casos estudados as localizações finais diferiam das localizações iniciais. Estes são os casos em que se pretenderam localizar 20 centros de serviço, tanto para 10 como para 100 cenários.

Ao continuar a avaliação da heurística apresentada, baseando-nos na mesma rede com 55 pontos de procura e para um total de 100 iterações, fixando o limite de tempo de espera no servidor (wlim), foram agora alterados os valores do limite de distância entre o servidor e o centro de procura (ldist).

Os valores testados para o limite de distância entre o servidor e o centro de procura foram ldist = 1, ldist = 10 e ldist = 20. Em todos os casos o limite de tempo de espera no servidor foi fixado em wlim = 20.

É precisamente para o estudo de ldist = 1, qualquer que seja o problema de localização proposto, que a heurística em análise funciona de tal modo que nos retribui sempre valores diferentes no que respeita as localizações finais e as localizações iniciais utilizadas como solução inicial.

Ao aumentar o limite de distância entre o servidor e o centro de procura para ldist = 10, 2 em 9 dos casos trabalhados apresentam diferença no que concerne as localizações iniciais e as localizações finais obtidas através da heurística. Tal sucede quando se pretende localizar 20 centros de serviço para os casos em que se simularam 10 e 100 cenários.

Por sua vez, quando se pretende utilizar ldist = 20, as conclusões são as mesmas, inclusivamente em termos do número de cenários, com a diferença que a não coincidência das localizações surge quando se pretende localizar 5 centros.

Tendo em consideração todos valores testados para o limite de tempo de espera (wlim = 0.02; 2; e 20 para ldist = 10) e para o limite de distância entre o servidor e o centro de procura (ldist = 1; 10; e 20 para wlim = 20), ao comparar o Regret Relativo Mínimo obtido para todos estes é possível constatar a sua tendência positiva face ao aumento do número de cenários testados.

No que respeita este último indicador, os resultados da heurística obtidos, relacionados com o aumento do número de centros de serviço a localizar, não apresentam um padrão confiável que nos permita uma generalização. Isto é possível de constatar tanto para a variação de ldist como para a variação de wlim.

4.5.2. Alterando a Dimensão da Rede e Número de Centros a Localizar.

Posteriormente, com o intuito de avaliar a heurística desenvolvida, foi conduzido um estudo onde se processaram dados obtidos simulando 100 exemplos e considerando em cada um destes a geração de 10 cenários. Desta feita, a dimensão da rede utilizada varia em termos do número de nós representativos dos pontos de procura. Optou-se assim por analisar redes com 25, 40 e 50 nós e, em cada um destes casos, variar o número de iterações. Consideram-se resultados obtidos para 500, 1000 e 2000 iterações.

No que respeita o tempo médio de processamento CPU (medido em segundos), é possível constatar que este aumenta devido a três fatores: o tamanho da rede, i.e., o número de nós de procura utilizados; o número de iterações; e, por último, o número de centros de serviço a localizar na resolução do problema. Contudo, este padrão não é tão linear quando se aprecia a

rede com 25 nós. Neste caso, para qualquer número de iterações considerado, ao se passar de 5 para 10 centros a localizar o tempo médio de processamento segue o padrão anteriormente descrito. Porém, ao passar de 10 para 20 centros de serviço a localizar, o tempo médio de processamento diminui.

Analisando a percentagem de localizações finais e iniciais coincidentes, estes valores são crescentes face ao número de nós na rede e face ao número de centros a localizar. Por sua vez, ao considerar um número crescente de iterações, constata-se, de um modo geral, que a percentagem de localizações coincidentes diminui. Os valores mais elevados para este indicador encontram-se nas redes mais pequenas e com menor número de iterações.

		Rede 25 Nós			**Rede 40 Nós**			**Rede 50 Nós**		
	Número de Centros a Localizar	***5***	***10***	***20***	***5***	***10***	***20***	***5***	***10***	***20***
500 iterações	*Tempo Médio de Processamento*	1,207	1,268	1,141	2,598	3,449	4,704	3,967	4,96	8,809
	Localizações Coincidentes %	5%	37%	95%	10%	24%	82%	14%	24%	92%
	Regret Médio	0,11866	0.15134	0,16351	0,08012	0,10989	0,12178	0,04482	0,08237	0.08504
1000 iterações	*Tempo Médio de Processamento*	1,661	2,227	1,806	3,645	5,661	8,242	5,742	8,015	13,88
	Localizações Coincidentes %	1%	21%	88%	5%	15%	77%	12%	22%	90%
	Regret Médio	0,1149	0,14312	0,16052	0,07705	0,10031	0,11625	0,06552	0,07983	0,08349
2000 iterações	*Tempo Médio de Processamento*	2,886	3,733	3,1	5,446	10,535	14,612	8,356	13,939	23,258
	Localizações Coincidentes %	5%	14%	84%	6%	7%	66%	18%	11%	64%
	Regret Médio	0.10729	0,13702	0,15885	0,06570	0,10122	0,12052	0,03293	0,06982	0,08623

Tabela 4.2. Resultados de simulação para 100 exemplos e 10 cenários; tempo médio de processamento medido em segundos.

4.6. Conclusões.

O aumento do limite para o tempo de espera (wlim), bem como o aumento da distância limite (ldist) entre o centro procura e o servidor, levam a crer que a heurística, analisando as seções anteriores, produz soluções que melhoram o padrão ótimo definido inicialmente.

Sob a perspetiva do congestionamento do sistema, menor número de centros a localizar e uma maior dimensão da rede proposta, levam a heurística a encontrar soluções diferentes das obtidas inicialmente como o método Regret.

Importa aqui analisar não só a percentagem de localizações coincidentes, mas também os valores do Regret Mínimo Relativo. Estes ao diminuírem indicam que soluções possíveis foram encontradas, sendo que estas, no caso de problemas para sistemas mais livres, se desviam menos da solução de partida para a heurística dada pelo Regret para os conjuntos de cenários simulados.

A mesma conclusão se pode retirar se equacionarmos os casos em que aumentaram o número de centros de serviço a localizar, levando-nos este aspeto a generalizar o comportamento da heurística desenvolvida.

Capítulo 5

Problema de Localização de Captura Máxima

Propomos agora um modelo alternativo que, apesar de se continuar a preocupar com a localização, pretende que a respetiva solução capte o máximo de quota de mercado dada a localização prévia de serviços patronizados por uma entidade concorrente. A secção 5.1. definirá o problema para posteriormente, em 5.2., se formular o Problema da Localização de Captura Máxima. Em 5.3. descreve-se o algoritmo proposto. Nas secções 5.4. e 5.5., respetivamente, apresentam-se os resultados da Experiência Computacional conduzida para o efeito e as Conclusões possíveis advenientes.

5.1. Definição do Problema.

Na maior parte dos exemplos que se pretendem estudar, além da distância entre o ponto de procura e o centro prestador do serviço, também o tempo de espera desempenha um papel importante na decisão final do consumidor. O número de elementos que determinado consumidor encontra em fila de espera deverá ser visto como uma medida da interpretação feita por este no que concerne o tempo que irá esperar pela sua vez de obter o serviço pretendido. Indo mais além, o tempo de espera despendido numa determinada "visita" poderá condicionar igualmente decisões futuras. Este tipo de problemas é verificado, e por nossas decisões solucionado, no nosso dia-a-dia ao recorrermos a determinada loja *franchise*, restaurantes *fast food* ou agências bancárias/máquinas de levantamento automático (ATM).

Em 1983, Kohlberg [56], num trabalho inédito desenvolvido com esta linha de raciocínio, constrói uma variante do modelo clássico de Hotelling [6] no que concerne a localização competitiva de lojas. Este autor assume que, aquando da escolha da loja, os consumidores consideram os fatores anteriormente enunciados apontando que o tempo de espera pelo serviço depende, em última análise, do número de consumidores que já haviam escolhido a mesma loja. Assumindo que cada um destes decide de modo a que a distância percorrida e o tempo de espera sejam minimizados, a quota de mercado será uma função contínua das respetivas localizações.

Por outro lado, no que respeitam as distâncias, é possível a generalização no sentido de as interpretar num contexto de funcionalidade, proximidade ou semelhança em vez de se fazer apenas uma apreciação geométrica da distância entre pontos. Em alguns tipos de serviço, o tempo de espera tem um impacto muito forte na perceção de proximidade tida pelo consumidor.

Neste enquadramento, o problema em estudo nesta seção tem como origem a estrutura seguida por ReVelle [11] quando desenvolveu o modelo de localização competitiva para uma determinada rede de procura: "*The Maximum Capture Problem*" (MAXCAP).

5.2. Formulação do Problema.

O modelo MAXCAP assume que, para um produto vendido homogéneo, a decisão do consumidor ao escolher uma determinada loja é baseada na distância. Assume-se ainda que os custos unitários para cada uma das lojas a localizar são idênticos. O modelo MAXCAP considera inicialmente uma empresa (empresa A), a qual irá competir com uma outra empresa (empresa B) que já opera no mercado com um número pré-determinado de servidores.

Na resolução deste tipo de problemas procura-se a localização de um número fixo de servidores (lojas) sob a tutela de um determinada empresa para um mercado onde já existem lojas concorrentes que competirão pelos mesmos clientes. O objetivo da empresa que agora entra neste mercado é o de maximizar os seus lucros. Sempre que os preços praticados entre as diferentes lojas são iguais e não são imputados custos diferentes em relação à opção verificada entre empresas concorrentes, o objetivo de maximização de lucro passa-se a traduzir em maximização das vendas totais.

Aqui, um consumidor pode ser o indivíduo, ou um grupo destes, com uma origem/localização e comportamento identificáveis. Como este pertence a um determinado local e realiza procura, recorre-se sem entraves ao termo ponto de procura. Cada consumidor revela uma atração específica em relação a cada uma das lojas concorrentes sendo que esta pode ser descrita por uma "função atração" (ou função utilidade).

Neste modelo MAXCAP os consumidores privilegiarão uma das empresas se o tempo de deslocação (medido em termos de distância *dij* for menor quando comparado com as lojas concorrentes.

Uma possível formulação para um problema MAXCAP é a seguinte:

$$Max\ \ Z^A = \sum_{i \in I} a_i Y_i^A + \sum_{i \in I} \frac{a_i}{2} Z_i \qquad (5.1)$$

sujeito a

$$Y_i^A \leq \sum_{j \in N_i(b_i)} X_j^A \qquad \forall i \in I \qquad (5.2)$$

$$Z_i \leq \sum_{j \in O_i(b_i)} X_j^A \qquad \forall i \in I \qquad (5.3)$$

$$Y_i^A + Z_i \leq 1 \qquad \forall i \in I \qquad (5.4)$$

$$\sum_{j \in J} X_j^A = p \qquad (5.5)$$

$$Y_i^A, Z_i, X_j^A \in \{0,1\} \qquad \forall i \in I, \forall j \in J$$

Sendo que:

- i, índice I e conjunto de ponto de procura;
- j, índice J e conjunto de localizações possíveis;
- a_i , procura no nó i;
- d_{ij} , distância do nó i ao nó j;
- b_i , servidor da empresa B mais perto do nó de procura i;
- d_{ibi} , distanciado nó de procura i ao servidor B mais próximo.

$$\mathrm{N_i}(b_i) = \{j \in J, d_{ij} < d_{ib_i}\}$$
$$O_i(b_i) = \{j \in J, d_{ij} = d_{ib_i}\}$$

$$Y_i^A = \begin{cases} 1, \text{caso a empresas A capture o nó de procura i} \\ 0, \text{caso contrário} \end{cases}$$

$$Z_i = \begin{cases} 1, \text{caso o nó de procura i seja dividido entre A e B} \\ 0, \text{caso contrário} \end{cases}$$

$$X_j^A = \begin{cases} 1, \text{caso a empresa A localize um servidor no nó j} \\ 0, \text{caso contrário} \end{cases}$$

O objetivo é o de maximizar a cobertura de mercado feita pela empresa A. A restrição (5.2) afirma que um nó de procura é coberto pela empresa A caso não exista um servidor da empresa B mais perto do referido nó de procura. Por seu turno, a restrição (5.3) considera que um nó de procura será dividido entre as duas empresas se a distância em relação ao servidor da empresa B mais próximo for a mesma que a distância ao servidor mais próximo da empresa A. A restrição (5.4) considera todos os 3 possíveis estados: a empresa A cobre o nó de procura, a empresa A e a empresa B ambas cobrem o nó de procura, ou nenhuma destas cobre nó de procura. Finalmente, a equação (5.5) correspondente à última restrição, determina o número de centros a localizar.

Kariv e Hakimi [57] provaram que este problema, tal como outros problemas p-mediano, é *NP-Hard*. Caso fosse a função objetivo não-linear, seria necessário resolver um modelo p-mediano para cada uma das possíveis localizações de uma loja da empresa A. Qualquer uma destas condicionantes, apesar de só se aplicar a primeira, justificam a utilização e a importância das meta-heurísticas oportunamente referidas.

5.3. Descrição do Algoritmo.

A descrição que se segue, tal como sugerido no problema estudado no capítulo anterior, enuncia todos os passos e procedimentos recursivos impostos através de um programa de computador cujo algoritmo, inspirado no p-minmax de Daskin [9], foi programado em C++ para atingir uma solução quase ótima para o problema.

Assim sendo, inicia-se com a leitura do ficheiro distância reconhecendo-se deste modo a rede de pontos de procura - todos eles potenciais localizações - e a distância entre os mesmos - nosso custo associado medido em termos de tempo.

É também nesta fase inicial de leitura de dados do problema que é obtida a informação relativa à empresa concorrente, cenário em relação ao qual pretendemos fazer a nossa localização. Para que tal aconteça, é necessário conhecer, em relação a cada uma das lojas da empresa concorrente, a sua distância em relação ao cliente/procura. Desta forma, temos a hipótese de, dada a caraterização da rede em termos de procura e localização das lojas da empresa concorrente, obter uma solução que pretenderá “encaixar” neste espaço as lojas da empresa em estudo de modo a cumprir o seu objetivo.

A caraterização de cada nó de procura em termos de população e frequência de procura por serviço segue um processo idêntico ao apresentado na seção 4.4. do Capítulo 4.

Sendo que o sistema foi inicializado, ou seja, temos em nossa posse uma caraterização da rede no que respeita o número de nós, distância entre estes, populações e procuras e localização das lojas concorrentes, recorrendo ao *software* de otimização CPLEX, o modelo começa por resolver o problema de localização com captura máxima para cada cenário.

O valor ótimo para a nossa função objetivo é então obtido passando a funcionar como futura referência aquando da comparação com os resultados da heurística. Esta solução inclui: quais os nós onde se localizam os centros de serviço e qual valor da solução objetivo no que concerne o total de população coberta/prevista.

Com base nesse padrão da solução ótima constrói-se uma matriz (ns, ns), que se intitula Regret 1, na qual cada linha e coluna representa um dos cenários simulados. Neste sentido, a diagonal dessas matrizes regret representa os valores ótimos obtidos em cada um dos cenários simulados. Os valores fora dessa diagonal, i.e., os valores adjacentes à solução ótima, são obtidos fazendo o cálculo da função objetivo considerando as caraterísticas dos outros cenários (populações e procuras) mas mantendo o padrão de localizações retribuídos pelo *software* de otimização CPLEX.

1669.05	1599	2068	1830	1754	1699	1767	1730	1929	1703
1545	1786.86	2109	1910	1788	1806	1882	1734	1956	1796
1587	1749	2138.88	1886	1765	1787	1881	1766	2032	1764
1545	1773	2109	1923.53	1788	1806	1882	1734	1956	1796
1555	1698	2057	1840	1830.25	1698	1761	1642	1894	1714
1584	1738	2115	1845	1789	1865.28	1947	1777	1958	1821
1584	1738	2115	1845	1789	1849	1960.12	1777	1958	1821
1642	1674	2120	1900	1728	1807	1888	1838.62	1991	1785
1587	1749	2124	1886	1765	1787	1881	1766	2045.97	1764
1566	1708	2065	1850	1792	1823	1921	1741	1938	1838.68

Figura 4. Matriz Regret 1 para 10 cenários.

Nos casos em que o valor objetivo calculado com o padrão de localizações para os restantes cenários for impossível, o valor que constará da matriz regret será zero sendo desta forma omitido da nossa heurística.

Seguindo um processo idêntico ao já enunciado no capítulo anterior (seção 4.4.) e com base na matriz Regret 1, constroem-se as matrizes Regret 2 e Regret 3, sendo que cada uma destas é uma transformação das anteriores.

0	-70.055	398.945	160.945	84.945	29.945	97.945	60.945	259.945	33.945
-241.855	0	322.145	123.145	114.499	19.145	95.145	-52.855	169.145	914.499
-551.885	-389.885	0	-252.885	-373.885	-351.885	-257.885	-372.885	-106.885	-374.885
-378.53	-150.53	185.47	0	-135.53	-117.53	-41.53	-189.53	32.47	-127.53
-275.25	-132.25	226.75	975.001	0	-132.25	-69.25	-188.25	63.75	-116.25
-281.285	-127.285	249.715	-20.285	-76.285	0	81.715	-88.285	92.715	-44.285
-376.12	-222.12	154.88	-115.12	-171.12	-111.12	0	-183.12	-212.001	-139.12
-196.615	-164.615	281.385	61.385	-110.615	-31.615	49.385	0	152.385	-53.615
-458.97	-296.97	78.03	-159.97	-280.97	-258.97	-164.97	-279.97	0	-281.97
-272.685	-130.685	226.315	11.315	-46.685	-15.685	82.315	-97.685	99.315	0

Figura 5. Matriz Regret 2 para 10 cenários.

É precisamente com base na matriz Regret 3 que o nosso algoritmo prossegue com a aplicação da heurística sugerida por Daskin et al. [9]. O método de escolha *Minmax* seguido pelo autor é idêntico ao oportunamente referido no capítulo 4 e permitirá obter o padrão de localização que funcionará como solução inicial para o posterior desenvolvimento do GRASP.

0,0000	0,0420	0,2390	0,0964	0,0509	0,0179	0,0587	0,0365	0,1557	0,0203
0,1354	0,0000	0,1803	0,0689	0,0006	0,0107	0,0532	0,0296	0,0947	0,0051
0,2580	0,1823	0,0000	0,1182	0,1748	0,1645	0,1206	0,1743	0,0500	0,1753
0,1968	0,0783	0,0964	0,0000	0,0705	0,0611	0,0216	0,0985	0,0169	0,0663
0,1504	0,0723	0,1239	0,0053	0,0000	0,0723	0,0378	0,1029	0,0348	0,0635
0,1508	0,0682	0,1339	0,0109	0,0409	0,0000	0,0438	0,0473	0,0497	0,0237
0,1919	0,1133	0,0790	0,0587	0,0873	0,0567	0,0000	0,0934	0,0011	0,0710
0,1069	0,0895	0,1530	0,0334	0,0602	0,0172	0,0269	0,0000	0,0829	0,0292
0,2243	0,1451	0,0381	0,0782	0,1373	0,1266	0,0806	0,1368	0,0000	0,1378
0,1483	0,0711	0,1231	0,0062	0,0254	0,0085	0,0448	0,0531	0,0540	0,0000

Figura 6. Matriz Regret 3 para 10 cenários.

O processo desenvolvido no GRASP é em tudo idêntico ao apresentado na seção 4.4. do capítulo anterior.

Em seguida, são novamente construídas as matrizes Regret 1, Regret 2 e Regret 3 a partir dos valores agora obtidos na *local search* realizada. Este processo é repetido para um número pré-definido de iterações.

5.4. Experiência Computacional.

Nesta seção é realizada uma experiência simples que consiste em gerar aleatoriamente situações distintas de forma a comparar os resultados da solução heurística com os resultados iniciais obtidos, ponto de partida para a fase de procura local da nossa heurística.

O problema de Localização com Captura Máxima explorado neste trabalho, tal como a heurística avaliada, baseiam-se no estudo de uma rede de pontos que será alterada no que respeita, principalmente, o número de nós ou centros de procura.

Neste contexto, no trabalho levado a cabo, são geradas redes com 25, 40, 50 nós e a cada um destes é atribuída a respetiva frequência de procura. Para cada nó, é gerada, de acordo com uma distribuição Uniforme [800;1800], a população a utilizar como dado no sentido de obter a frequência de procura. Dada a população, 1% é considerada a frequência de procura (ou necessidade de serviço/atendimento).

Note-se que, em cada cenário gerado e para rede específica, a distância entre nós será constante dado que se altera apenas o tamanho da rede e a procura verificada em cada ponto da mesma. A distância entre pontos de procura é obtida com recurso a uma matriz distâncias comum a todos

os cenários e redes estudadas, sucedendo o mesmo para a informação de distâncias relativa à empresa concorrente sendo esta conhecida à *priori* da nossa solução de localização.

A tabela seguinte apresenta um resumo das caraterísticas e parâmetros dos conjuntos de dados trabalhados.

	Número		**Número**	
Casos	**Nós**	**Centros**	**Iterações**	**Cenários**
1	50	5, 10 e 20	500, 1000 e 2000	10
2	40	5, 10 e 20	500, 1000 e 2000	10
3	25	5, 10 e 20	500, 1000 e 2000	10

Tabela 5.1. Caraterísticas e parâmetros dos conjuntos de dados processados

O algoritmo em estudo foi implementado num computador com processador Dual 2.50 GHz Pentium Dual-Core com 1920 MB de memória e recorrendo ao compilador C++ Microsoft Visual Studio 2005 onde se integra, para a resolução dos problemas propostos, o *software* de otimização CPLEX *Optimization Studio* 12.2.

O nosso objetivo ao analisar os resultados obtidos é verificar até que ponto o padrão de localizações, retribuídos pela heurística, é significativamente diferente do padrão ótimo de localizações obtido na resolução do problema.

5.4.1. Alterando a Dimensão da Rede e Número de Centros a Localizar

Tendo em vista a apreciação da heurística desenvolvida para o problema em estudo neste capítulo, foram analisados os dados que a seguir se apresentam na Tabela 5.2.

		Rede 25 Nós			Rede 40 Nós			Rede 50 Nós		
	Número de Centros a Localizar	***5***	***10***	***20***	***5***	***10***	***20***	***5***	***10***	***20***
500 iterações	*Tempo Médio de Processamento*	4.955	20.73	84.01	12.11	51.1	208.35	19.06	81.84	345.62
	Localizações Coincidentes %	100%	95%	81%	100%	99%	99%	100%	100%	100%
	Regret Médio	0.18213	0.16835	0.18075	0.14728	0.1164	0.13884	0.13837	0.13418	0.12164
1000 iterações	*Tempo Médio de Processamento*	10.14	41.35	183.17	24.08	101.9	432.69	37.25	159.29	670.45
	Localizações Coincidentes %	95%	99%	73%	100%	99%	99%	71%	100%	80%
	Regret Médio	0.18373	0.15902	0.17961	0.12047	0.117	0.12721	0.11058	0.12614	0.11036
2000 iterações	*Tempo Médio de Processamento*	20.12	82.47	337.36	47.83	202.39	830.34	81.96	325.31	1291.7
	Localizações Coincidentes %	100%	87%	81%	100%	100%	98%	95%	100%	55%
	Regret Médio	0.18284	0.18278	0.18304	0.16338	0.14242	0.11598	0.13074	0.14418	0.11701

Tabela 5.2. Resultados de simulação para 100 exemplos e 10 cenários; tempo médio de processamento medido em segundos.

Como se constata, variou a dimensão da rede utilizada - 25, 40 e 50 nós – e o número de centros a localizar – 5, 10 e 20 – para qualquer número de iterações definidas. Em todos os casos processaram-se dados obtidos simulando 100 exemplos e considerando em cada um destes a geração de 10 cenários.

O tempo médio de processamento CPU (em segundos), talvez dos indicadores mais sensíveis no estudo desenvolvido no presente capítulo, verifica-se que aumenta exponencialmente com o aumento da dimensão da rede, i.e., o número de nós de procura considerados. Este indicador apresenta o mesmo tipo de comportamento no que respeita a sua relação com o número de iterações e com o número de centros de serviço a localizar na resolução do problema. Em qualquer um destes casos, perante o aumento da dimensão da rede, do número de centros a localizar e do número de iterações, o tempo de processamento apresenta o comportamento indicado.

Quanto à percentagem de localizações finais e iniciais coincidentes, estes valores são crescentes face ao número de nós na rede e face ao número de centros a localizar. Por sua vez, ao considerar um número crescente de iterações, constata-se, de um modo geral, que a percentagem de localizações coincidentes diminui. Os valores mais elevados para este indicador encontram-se nas redes mais pequenas e com menor número de iterações.

Se bem que apresente um comportamento não linear, é possível verificar que, com o aumento da dimensão da rede proposta, o valor do Regret Relativo Mínimo diminui.

5.5. Conclusões.

Aplicando a mesma heurística, mas neste capítulo ao Problema de Captura Máxima, é possível constatar que o menor congestionamento do sistema permite a heurística melhorar a solução inicial obtida através da aplicação do procedimento Regret. Nota-se essencialmente que, para casos com maiores espaços de soluções, apesar do aumento no tempo de processamento, a heurística produz soluções diferentes da solução inicial. Incluímos neste sentido o aumento do número de centros a localizar bem como, se bem que menos generalizável, o aumento da dimensão da rede proposta.

Capítulo 6

Problema de Localização de Infraestruturas com Capacidade Limitada

Este capítulo propõe um modelo de minimização de custos baseado no problema "*Facility Location Problem*". A secção 6.1. define o problema e as suas várias abordagens e em 6.2. apresenta-se a formulação para o mesmo. Após na secção 6.3. se descrever o algoritmo utilizado, em 6.4. e 6.5. respetivamente, conduz-se a Experiência Computacional para depois serem tiradas algumas Conclusões.

6.1. Definição do Problema.

Neste estudo, os custos relacionados com o transporte e os custos fixos específicos relacionados com cada localização pretendida são, de uma forma geral, a principal condicionante na aferição do preço de determinado bem. O "*Facility Location Problem*" (FLP), introduzido por Balinski [8], aborda o caso de localizar um novo conjunto de infraestruturas que providenciam serviços de tal modo que a soma dos dois custos mencionados anteriormente seja mínima.

Um caso mais específico, denominado "*Single Source Capacitated Plant Location*", de estrutura p-mediano, na tentativa de incorporar outros elementos que possam refinar a decisão final, introduz outros fatores:

i) associa um custo fixo que será diferente consoante cada um dos locais potenciais considerados; e

ii) cada uma das instalações que estão sendo localizadas deve possuir cabimentação, i.e., há restrições no que respeita a capacidade de cada uma destas.

É precisamente em relação a este último fator que é dado agora especial atenção. Ao "desenhar" um sistema de produção com diversas infraestruturas deve-se ter em conta a capacidade do mesmo, dadas as caraterísticas das instalações, tendo em conta satisfazer todas as ordens de procura.

6.2. Formulação do Problema.

O Problema de Localização de Infraestruturas com Capacidade Limitada é uma versão do Problema de Localização de Infraestruturas. O modelo original não considerava a capacidade de prestação de serviço das instalações localizadas sendo que agora, com a sua inclusão, a localização de custo mínimo poderá não ser capaz de atender a totalidade de procura verificada.

Este problema tem como objetivo a minimização do custo total da instalação, bem como a minimização dos custos relacionados com o transporte perante as distâncias percorridas, sendo a formalização a que se apresenta a seguir:

$$Min \qquad \sum_{j\in J} f_j X_j + \sum_{i\in I}\sum_{j\in J} d_{ij} Y_{ij} \qquad (6.1)$$

sujeito a

$$\sum_{j\in J} Y_{ij} = 1 \qquad \forall i \in I \qquad (6.2)$$

$$Y_{ij} - X_j \le 0 \qquad \forall i \in I, \forall j \in J \qquad (6.3)$$

$$\sum_{i\in I} a_i Y_{ij} \le C_j \qquad \forall j \in J \qquad (6.4)$$

$$X_j \in \{0,1\},\ Y_{ij} \in \{0,1\} \quad \forall i \in I, \forall j \in J$$

Na formulação apresentada $I = \{1,...,m\}$ representa um conjunto de clientes com procura f_i, $i \in I$, que será servido pelas infraestruturas localizadas num subconjunto de nós do total $J = \{1,...,n\}$ de potenciais localizações. Para cada localização $j \in J$, o custo fixo de operar a infraestrutura em *j* é f_j e a respetiva capacidade é C_j. O custo, em termos de distância percorrida, de afetar a instalação *j* ao cliente *i* é d_{ij}.

A restrição (6.2) força cada ponto de procura a ser atribuído apenas um centro. A restrição apresentada em (6.3) obriga a que a afetação seja feita apenas a infraestruturas que se encontram a operar, i.e., abertas. Por fim, a restrição (6.4), conhecida como a restrição de capacidade, limita a população total atribuída a uma instalação de acordo com a respetiva capacidade de instalação.

O problema de localização com capacidades incluídas pode ser descrito como se segue: há um conjunto de potenciais localizações para as instalações consideradas aos quais correspondem custos fixos e respetivas capacidades; há um conjunto de consumidores com uma determinada procura pelos bens disponibilizados pelas instalações mencionadas. É conhecido o custo de transporte por cada unidade do bem fornecido.

A questão agora prende-se com a determinação de um subconjunto dessas instalações que minimize o total de custos fixos e custos relacionados com o transporte de modo a satisfazer toda a procura sem violar as restrições de capacidade impostas. Este problema desenvolve-se em 2 estágios: encontrar o subconjunto de instalações a serem localizadas e afetar a estas os respetivos consumidores.

Quando se incorpora a restrição (6.2) obtém-se a versão "*Single Source*" do problema de localização na medida em que determinada procurada será afetada apenas a uma determinada infraestrutura. Uma outra hipótese presente neste problema considera que os custos de transporte são linearmente dependentes da quantidade transportada não podendo ser verificadas, neste sentido, economias de escala. São diversas as referências a este problema e, para efeitos de revisão, deve-se consultar como exemplo Sridharan [58].

6.3. Descrição do Algoritmo.

A descrição que se segue, tal como sugerido no problema estudado nos capítulos anteriores, pretende enunciar os procedimentos recursivos impostos através de um programa de computador cujo algoritmo, inspirado no p-minmax de Daskin [9], foi programado em C++ para atingir uma solução quase ótima para o problema.

O sistema inicia do mesmo modo que os anteriores, precisamente com a leitura de um ficheiro distância de modo a reconhecer a rede de pontos de procura proposta, sendo que todos estes pontos serão também potenciais localizações. Dada a estrutura da rede é também associada uma distância entre todos os nós.

Note-se que o objetivo deste problema, não deixando de se preocupar com uma solução em termos de número de infraestruturas a localizar e sua respetiva "posição" na referida rede, se prende com a minimização de custos. Desta forma, além de incluir na função objetivo as distâncias percorridas que funcionam como custos de transporte associados a uma dada localização ($\sum_{i\epsilon I}\sum_{j\epsilon J} d_{ij}Y_{ij}$ da função objetivo (6.1)), será necessário adicionalmente atribuir a cada infraestrutura o respetivo custo fixo ($\sum_{j\epsilon J} f_j X_j$ da função objetivo (6.1)).

Neste sentido, o programa segue lendo um ficheiro denominado "*f j*" que associa a cada ponto de procura e potencial localização um custo de modo a que se solucione o problema minimizando então o somatório dos custos relacionados com distâncias percorridas e custo fixos (equação (6.1)).

A caraterização de cada nó de procura em termos de população e frequência de procura por serviço segue um processo idêntico ao já apresentado nas seções 4.4. do Capítulo 4 e 5.3 do Capítulo 5.

Sendo que o sistema foi inicializado, ou seja, temos em nossa posse uma caraterização da rede no que respeita o número de nós, distância entre estes, populações e procuras e respetivos custos fixos, recorrendo ao *software* de otimização CPLEX, o modelo começa por resolver o problema de localização de infraestruturas com capacidade limitada para cada cenário.

Tenha-se em atenção que a capacidade de cada uma destas é um valor definido à priori e que, pelo facto de vir a assumir diferentes valores durante a nossa experiência, funcionará como ferramenta para a obtenção de resultados passíveis de uma análise no que respeita a resposta do sistema face aos seus vários estados. Os valores que utilizaremos para a capacidade durante o estudo levado a cabo serão posteriormente apresentados na seção 6.4 deste capítulo.

O valor ótimo para a nossa função objetivo é assim obtido passando a funcionar como futura referência aquando da comparação com os resultados da heurística. Esta solução inclui quais os nós onde se localizam os centros de serviço e, assim sendo, quantos são os centros de serviço que serão localizados. Conhecendo também nesta solução a afetação dos pontos de procura às localizações definidas, determina-se a solução objetivo no que concerne um total de custos que engloba tanto custos fixos como custos relacionados com distâncias percorridas.

Com base nesse padrão da solução ótima constrói-se uma matriz (ns, ns), que se intitula Regret 1, na qual cada linha e coluna representa um dos cenários simulados. Neste sentido, a diagonal dessas matrizes regret representa os valores ótimos obtidos em cada um dos cenários simulados. Os valores fora dessa diagonal, i.e., os valores adjacentes à solução ótima, são obtidos fazendo o cálculo da função objetivo considerando as caraterísticas dos outros cenários (populações e procuras) mas mantendo o padrão de localizações e afetações retribuídos pelo *software* de otimização CPLEX.

946.5	1555	1823	1859	1749	1246	1715	1478	1350	1650
1377	951.02	1345	1720	1617	1711	1933	1695	1121	1346
1459	1768	975.31	1650	1919	1373	1166	1180	1813	1889
1779	1538	1403	910.28	1661	1586	1645	1587	1632	1452
1582	1556	1910	1505	910.4	1385	1197	1595	1578	1452
1271	1419	1779	1989	2172	889.79	1695	1408	1676	1461
1577	1425	1209	1198	2163	1102	881.35	1283	1689	1661
1594	1406	1766	1773	1917	1590	1429	972.26	1383	1886
1143	1549	1760	2060	1415	2045	2012	1733	960.34	1631
1549	1425	2027	1444	1906	1403	1349	1483	1536	923.25

Figura 7. Matriz Regret 1 para 10 cenários.

Nos casos em que o valor objetivo calculado com o padrão de localizações para os restantes cenários for impossível, o valor que constará da matriz regret será zero sendo desta forma omitido da nossa heurística.

Seguindo um processo idêntico ao já enunciado nos capítulos anteriores (seção 4.4. e seção 5.3) e com base na matriz Regret 1, constroem-se as matrizes Regret 2 e Regret 3, sendo que cada uma destas é uma transformação das anteriores.

0	608.5	876.5	912.5	802.5	299.5	768.5	531.5	403.5	703.5
425.98	0	393.98	768.98	665.98	759.98	981.98	743.98	169.98	394.98
483.69	792.69	0	674.69	943.69	397.69	190.69	204.69	837.69	913.69
868.72	627.72	492.72	0	750.72	675.72	734.72	676.72	721.72	541.72
671.6	645.6	999.6	594.6	0	474.6	286.6	684.6	667.6	541.6
381.21	529.21	889.21	1099.21	1282.21	0	805.21	518.21	786.21	571.21
695.65	543.65	327.65	316.65	1281.65	220.65	0	401.65	807.65	779.65
621.74	433.74	793.74	800.74	944.74	617.74	456.74	0	410.74	913.74
182.66	588.66	799.66	1099.66	454.66	1084.66	1051.66	772.66	0	670.66
625.75	501.75	1103.75	520.75	982.75	479.75	425.75	559.75	612.75	0

Figura 8. Matriz Regret 2 para 10 cenários.

Com base na matriz Regret 3 o nosso algoritmo prossegue aplicando-se a heurística sugerida por Daskin et al. [9]. O método de escolha *Minmax* seguido pelo autor é idêntico ao oportunamente referido nos capítulos 4 e 5 e permitirá obter o padrão de localização e afetação que funcionará como solução inicial para o posterior desenvolvimento do GRASP.

0	0.642895	0.926043	0.964078	0.847861	0.316429	0.811939	0.561543	0.426307	0.743265
0.447919	0	0.414271	0.808584	0.70028	0.799121	103.255	0.782297	0.178734	0.415322
0.495935	0.812757	0	0.69177	0.96758	0.407758	0.195517	0.209872	0.858896	0.93682
0.954344	0.68959	0.541284	0	0.824713	0.742321	0.807136	0.74342	0.792855	0.595114
0.737698	0.709139	109.798	0.65312	0	0.521309	0.314807	0.751977	0.733304	0.594903
0.428427	0.594758	0.999348	123.536	144.103	0	0.904944	0.582396	0.883591	0.64196
0.789301	0.616838	0.371759	0.359278	145.419	0.250355	0	0.455721	0.916378	0.884609
0.639479	0.446115	0.816387	0.823586	0.971695	0.635365	0.469771	0	0.422459	0.93981
0.190203	0.61297	0.832684	114.507	0.473436	112.945	109.509	0.804569	0	0.698357
0.677769	0.543461	119.551	0.56404	106.445	0.519632	0.461143	0.606282	0.663688	0

Figura 9. Matriz Regret 3 para 10 cenários.

O processo desenvolvido no GRASP é em tudo idêntico ao apresentado nas seções 4.4. e 5.3 dos anteriores capítulos.

De seguida, são novamente construídas as matrizes Regret 1, Regret 2 e Regret 3 a partir dos valores agora obtidos na *local search* realizada. Este processo é repetido para um número pré-definido de iterações.

6.4. Experiência Computacional.

Tal como desenvolvido em capítulos anteriores, com o objetivo observar a diferença entre os resultados da solução heurística e a solução inicial possível para o problema, foi realizada uma experiência que consiste em gerar aleatoriamente situações problemáticas de forma a comparar os resultados da solução heurística com os resultados iniciais obtidos que funcionam como ponto de partida para o processo heurístico desenvolvido.

O problema de Localização de Infraestruturas com Capacidade Limitada explorado, tal como a heurística avaliada, baseiam-se no estudo de uma rede de pontos que representam centros de procura e potenciais localizações. A dimensão desta rede será variável e a cada nó será atribuída uma determinada frequência de procura (necessidade de serviço/atendimento).

As caraterísticas desta rede podem ser alteradas no que respeita, principalmente, o número de nós ou centros de procura. Serão também alteradas as capacidades associadas a cada uma das infraestruturas que se irão localizar.

Neste contexto, serão geradas redes com 25, 40, 50 nós e a cada um destes é atribuída a respetiva frequência de procura. Para cada nó, é gerada, de acordo com uma distribuição Uniforme [800;1800], a população a utilizar como dado no sentido de obter a frequência de procura. Dada a população, 1% é considerada a frequência de procura (ou necessidade de serviço/atendimento).

Ter em atenção que, para cada cenário gerado e rede específica, a distância entre nós será constante dado que se altera apenas o tamanho da rede e a procura verificada em cada ponto da mesma. A distância entre pontos de procura é obtida com recurso a uma matriz distâncias comum a todos os cenários e redes estudadas.

Tendo como objetivo testar a heurística desenvolvida, além das diferenças de dados no que concerne a dimensão da rede, foram também fixados valores alternativos para a capacidade das infraestruturas, como se poderá verificar na Tabela 6.1.

No que respeita o processo recursivo do algoritmo, tendo em vista o apuramento da qualidade das conclusões, foram testados programas ora com um número diferente de iterações, ora com capacidades de processamento diferentes, tal como se apresenta na próxima tabela.

Casos	**Número Nós**	**Limites Capacidade**	**Número**	
			Cenários	**Iterações**
1	50	1000, 2000 e 3000	10	500, 1000 e 2000
2	40	1000, 2000 e 3000	10	500, 1000 e 2000
3	25	1000, 2000 e 3000	10	500, 1000 e 2000

Tabela 6.1. Caraterísticas e parâmetros dos conjuntos de dados processados.

O algoritmo em estudo foi implementado num computador com processador Dual 2.50 GHz Pentium Dual-Core com 1920 MB de memória e recorrendo ao compilador C++ Microsoft Visual Studio 2005 onde se integra, para a resolução dos problemas propostos, o *software* de otimização CPLEX *Optimization Studio* 12.2.

Em termos gerais, o nosso objetivo ao analisar os resultados obtidos é verificar até que ponto o padrão de localizações e afetações, retribuídos pela heurística, mostra haver ou não diferenças em relação ao padrão ótimo de localizações e afetações obtido na resolução do problema.

6.4.1. Alterando a Dimensão da Rede

Na presente seção, pretende-se analisar a resposta da heurística desenvolvida, quando comparada com a solução inicial, no que respeita, além dos indicadores apreciados nos capítulos anteriores, o número de centros localizados. Neste sentido, passamos a generalizar o comportamento do modelo ao aumentar a dimensão da rede e ao aumentar o número de iterações, tal como apresentado na tabela 6.1.

O tempo de processamento (em segundos), perante o aumento do número de nós da rede e dado um número crescente de iterações, tem patente um comportamento crescente. Se considerarmos apenas a dimensão da rede, este aumento do tempo de processamento é exponencial.

Como seria de esperar, também com o aumento da rede considerada, o número de centros localizados aumenta e, de um modo geral, o mesmo acontece ao valor do Regret Relativo Mínimo. Neste caso, se se pretender comparar a solução inicial com a solução heurística, verifica-se existirem casos de localizações iniciais e finais coincidentes, se bem que com um padrão de certa forma errático.

No que respeita soluções coincidentes, o aumento do número de iterações não permite igualmente generalizar um padrão comportamental deste nosso indicador. É possível porém, verificar alguma estabilidade no comportamento dos valores Regret Relativo Mínimo. Uma chamada de atenção contudo para uma exceção que se refere a uma

diminuição deste indicador perante a apresentação de um modelo mais "apertado" com capacidades mais reduzidas e uma rede de maior dimensão.

		Rede 25 Nós			Rede 40 Nós			Rede 50 Nós		
	Capacidade	***1000***	***2000***	***3000***	***1000***	***2000***	***3000***	***1000***	***2000***	***3000***
500 iterações	*Tempo Médio de Processamento*	3.399	0,738	0.756	68.26	4.115	1.784	119.41	5.611	4.778
	Localizações Coincidentes %	0%	10%	10%	0%	0%	16%	0%	0%	2%
	Regret Médio	1.417	4.970	0.831	1.563	1.201	1.088	1.399	1.112	1.290
	Nº Centros Localizados	2	1	1	3	2	1	3	2	2
1000 iterações	*Tempo Médio de Processamento*	4.101	1.042	1.001	72.379	5.471	2.443	137.42	7.138	6.532
	Localizações Coincidentes %	6%	2%	12%	0%	4%	16%	0%	4%	10%
	Regret Médio	1.443	2.809	0.833	1.579	1.299	1.102	1.601	1.191	1.282
	Nº Centros Localizados	2	1	1	3	2	2	3	2	2
2000 iterações	*Tempo Médio de Processamento*	5.798	1.453	1.570	69.357	8.521	3.461	233.82	12.10	11,839
	Localizações Coincidentes %	0%	4%	8%	0%	6%	8%	0%	2%	6%
	Regret Médio	1.423	0.832	0.844	1.565	1.117	1.043	1.599	1.219	1.259
	Nº Centros Localizados	2	1	1	3	2	1	3	2	2

Tabela 6.2. Resultados de simulação para 100 exemplos e 10 cenários; tempo médio de processamento medido em segundos.

6.4.2. Alterando as Capacidades

Analisando os dados já apresentados na tabela 6.2, agora nesta seção no que respeita os valores para a capacidade de processamento da procura pelas infraestruturas localizadas.

O aumento da capacidade de serviço permite descongestionar o sistema e é possível concluir isso ao verificar que o número de centros localizados diminui. Como seria de esperar, neste caso, pela simplicidade do problema proposto, igualmente o tempo de processamento decresce para as capacidades maiores.

Por último, aquando do aumento das referidas capacidades inerentes a cada infraestrutura localizada, a diminuição dos valores Regret em conjunto com a crescente percentagem de localizações iniciais e finais coincidentes, deixam antever que problemas mais simples (ou modelos menos congestionados) permitem a heurística melhorar a solução inicial apontada.

6.5. Conclusões.

Aplicando a mesma heurística ao Problema de Localização de Infraestruturas com Capacidades Limitadas, são possíveis de constatar precisamente as mesmas conclusões que as obtidas no capítulo 4. Nota-se essencialmente que para casos em que existe maior congestionamento, apesar do aumento no tempo de processamento, a heurística produz soluções diferentes da solução inicial obtida pelo procedimento Regret mascom desvios relativos menores em relação a essa solução.

O “aperto” provocado ao sistema, no problema em causa, prende-se principalmente com as caraterísticas das infraestruturas a localizar no que concerne as suas capacidades de processamento/atendimento da procura.

Capítulo 7

Conclusões

Ao analisar a literatura relacionada com problemas de localização e afetação, nota-se a tendência de incluir neste tipo de modelos os efeitos das filas de espera. Tal sucede uma vez que, ao considerar determinada procura de serviço, na realidade, constata-se que esta é aleatória e uma das fontes de congestionamento dos sistemas. É por esta razão que o presente trabalho associa ao modelo de Localização de Cobertura Máxima formulações relacionadas com a teoria de filas de espera.

Este tipo de problemas, que tanto podem surgir no contexto do setor público ou privado, implicam tipos de formulação distintos como modelos de distância máxima ou modelos de distância total/média. A metodologia associada a cada problema específico deve ser proposta tendo esse cuidado e devendo-se comparar os resultados obtidos com outros atingidos por modelos tradicionais.

Adicionalmente ao "*Greedy Randomized Adaptive Search Procedure*" (ou GRASP), foi também utilizada na heurística desenvolvida o método p-minmax Regret (Regret Relativo Mínimo) proposto por Daskin [9]. A utilização desses processos vem na linha de pesquisas anteriores e visa a sua integração com o intuito de explorar novas metodologias que melhorem ou melhor se adaptem às circunstâncias dos casos estudados. Desta forma, os modelos desenvolvidos podem-se considerar adequados à resolução do tipo de problema proposto no corrente trabalho. Fazendo variar os limites em termos de tempos de espera e de distância máxima percorrida, limites de capacidades de processamento da procura e dimensão das redes, é possível verificar mudanças significativas nas soluções finais.

Os problemas em estudo são o bem conhecido Problema de Localização com Cobertura Máxima (Capítulo 4) de ReVelle [11] e um modelo alternativo no qual o comportamento de escolha do servidor não depende apenas do tempo percorrido do nó ao centro, mas também inclui o tempo de espera pelo serviço. Posteriormente (Capítulos 5 e 6 respetivamente) foram abordados o Problema de Localização de Captura Máxima, do mesmo autor, e o Problema de Localização de Infraestruturas com Capacidades Limitadas com base no problema estudado originalmente por Balinski [8].

São inúmeras as situações reais em que o tempo de espera é um dos fatores importantes ao considerar o tempo de serviço (tempo ou distância percorrida mais o tempo de espera). Nestas, tendo em conta a determinação de um padrão de localização, o tempo de espera é de considerar como indispensável na respetiva modelação. Podem igualmente interferir no tempo de processamento o número de centros a localizar perante a dimensão da rede, bem como a capacidade das instalações em providenciarem o serviço procurado.

A meta-heurística proposta retribui resultados próximos do ótimo demonstrando poupança significativa em termos de tempo de computação. Tendo em conta os dados iniciais, foi com o recurso à simulação que no presente trabalho se obtêm os níveis de procura associados a cada dado populacional.

No que concerne a aplicação das heurísticas *Greedy* a estas formulações, estas apresentam um comportamento aceitável na medida em que as soluções quase-ótimas se mostram sensíveis aos exemplos trabalhados e às situações problemáticas propostas em cada caso.

Por outro lado, a teoria estudada, bem como os exemplos numéricos obtidos, sugerem que as soluções se tornam menos sensíveis aos parâmetros do modelo quanto menos congestionado

estiver o sistema. No caso em que, por exemplo, o limite de distância entre o nó de procura, o centro servidor é menor ou existem menos centros a localizar, pode-se admitir haver agora um maior congestionamento associado ao modelo. Estes últimos são precisamente os casos em que a heurística apresenta soluções não idênticas ao ótimo.

Uma chamada de atenção porém para o Problema de Localização de Captura Máxima que apresenta conclusões contrárias às elaboradas para os outros dois problemas em estudo. Quer-se com isto dizer que a heurística produz resultados diferentes do ótimo perante o menor congestionamento do sistema.

Quanto à generalização das conclusões no que concerne a experiência computacional levada a cabo, deve ser tido algum cuidado especial. Os modelos testados e os seus vários exemplos foram obtidos com recurso à geração numérica aleatória. Em muitos casos, evidenciam-se resultados distintos mas existem outros onde a formulação proposta não produz diferenças significativas nos resultados. Como já foi oportunamente referido, de uma forma geral, nos sistemas mais "apertados", ou seja, onde o limite de distância seja mais pequeno, o número de centros de serviço sejam em menor número ou as capacidades de processamento das infraestruturas menores, as decisões de localização são mais sensíveis aos parâmetros pré-definidos para o modelo.

Em conclusão, tendo-se simulado as populações e as respetivas frequências de procura, com este trabalho consegue-se evidenciar a suma importância, tal como na vida real, de considerar o congestionamento dos sistemas nas suas várias vertentes como um fator determinante nas decisões de localização e afetação.

Referências Bibliográficas.

1. Current, J., M. Daskin, and D. Schilling, *3 D iscrete N etw ork Location M odels.* 2002.
2. Hakimi, S.L., *Optimum locations of switching centers and the absolute centers and medians of a graph.* Operations Research, 1964: p. 450-459.
3. Hakimi, S., *Optimum distribution of switching centers in a communication network and some related graph theoretic problems.* Operations Research, 1965. **13**(3): p. 462-475.
4. Marianov, V. and D. Serra, *Probabilistic, maximal covering location-allocation models for congested systems.* Journal of Regional Science, 1998. **38**(3): p. 401-424.
5. Silva, F. and D. Serra, *Locating emergency services with different priorities: the priority queuing covering location problem.* Journal of the Operational Research Society, 2008. **59**(9): p. 1229-1238.
6. Hotelling, H., *Stability in competition.* The economic journal, 1929. **39**(153): p. 41-57.
7. ReVelle, C. and K. Hogan, *The maximum availability location problem.* Transportation Science, 1989. **23**(3): p. 192-200.
8. Balinski, M.L., *Integer programming: methods, uses, computations.* Management Science, 1965. **12**(3): p. 253-313.
9. Daskin, M.S., S.M. Hesse, and C.S. Revelle, *α-reliable p-minimax regret: A new model for strategic facility location modeling.* Location Science, 1997. **5**(4): p. 227-246.
10. Friesz, T.L., T. Miller, and R.L. Tobin, *Competitive Network Facility Location Models: A Survey.* Papers in Regional Science, 1988. **65**(1): p. 47-57.
11. ReVelle, C., *The Maximum Capture or "Sphere of Influence" Location Problem: Hotelling Revisited on a Network*.* Journal of Regional Science, 1986. **26**(2): p. 343-358.
12. Toregas, C., et al., *The location of emergency service facilities.* Operations Research, 1971. **19**(6): p. 1363-1373.
13. Church, R. and C.Revelle, *The maximal covering location problem.* Papers in Regional Science, 1974. **32**(1): p. 101-118.
14. Kuby, M.J., *Programming Models for Facility Dispersion: The p-Dispersion and Maxisum Dispersion Problems.* Geographical Analysis, 1987. **19**(4): p. 315-329.
15. Hogan, K. and C. Revelle, *Concepts and applications of backup coverage.* Management Science, 1986. **32**(11): p. 1434-1444.
16. Daskin, M.S., *Application of an Expected Covering Model to Emergency Medical Service Sustem Design*.* Decision Sciences, 1982. **13**(3): p. 416-439.
17. Daskin, M.S., *A Maximum Expected Covering Location Model - Formulation, Properties and Heuristic Solution.* Transportation Science, 1983. **17**(1): p. 48-70.
18. Tan, K.C., C.Y. Cheong, and C.K. Goh, *Solving multiobjective vehicle routing problem with stochastic demand via evolutionary computation.* European Journal of Operational Research, 2007. **177**(2): p. 813-839.
19. Ozdemir, D., E. Yucesan, and Y.T. Herer, *Multi-location transshipment problem with capacitated production and lost sales.* Proceedings of the 2006 Winter Simulation Conference, Vols 1-5, 2006: p. 1470-1476.
20. Larson, R.C., *A hypercube queuing model for facility location and redistricting in urban emergency services.* Computers & Operations Research, 1974. **1**(1): p. 67-95.
21. Batta, R., *A queueing-location model with expected service time dependent queueing disciplines.* European Journal of Operational Research, 1989. **39**(2): p. 192-205.
22. Marianov, V. and C. Revelle, *The Queuing Probabilistic Location Set Covering Problem and Some Extensions.* Socio-Economic Planning Sciences, 1994. **28**(3): p. 167-178.
23. Berman, O., R.C. Larson, and S.S. Chiu, *Optimal server location on a network operating as an M/G/1 queue.* Operations Research, 1985. **33**(4): p. 746-771.
24. Batta, R. and N.R. Mannur, *Covering-location models for emergency situations that require multiple response units.* Management Science, 1990: p. 16-23.
25. Ball, M.O. and F.L. Lin, *A reliability model applied to emergency service vehicle location.* Operations Research, 1993: p. 18-36.
26. Mandell, M.B., *Covering models for two-tiered emergency medical services systems.* Location Science, 1998. **6**(1-4): p. 355-368.

27. Jamil, M., A. Baveja, and R. Batta, *The stochastic queue center problem.* Computers & Operations Research, 1999. **26**(14): p. 1423-1436.
28. Berman, O. and S. Vasudeva, *Approximating performance measures for public services.* Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 2005. **35**(4): p. 583-591.
29. Branas, C.C. and C.S. Revelle, *An iterative switching heuristic to locate hospitals and helicopters.* Socio-Economic Planning Sciences, 2001. **35**(1): p. 11-30.
30. Harewood, S., *Emergency ambulance deployment in Barbados: a multi-objective approach.* Journal of the Operational Research Society, 2002. **53**(2): p. 185-192.
31. Marianov, V. and D. Serra, *4 Location Problems in the Public Sector.* 2002.
32. Chiu, S.S., O. Berman, and R.C. Larson, *Locating a mobile server queueing facility on a tree network.* Management Science, 1985: p. 764-772.
33. Batta, R., R.C. Larson, and A.R. Odoni, *A single-server priority queueing-location model.* Networks, 1988. **18**(2): p. 87-103.
34. Batta, R. and O. Berman, *A Location Model for a Facility Operating as a M/G/K Queue.* Networks, 1989. **19**(6): p. 717-728.
35. Brandeau, M.L. and S.S. Chiu, *A center location problem with congestion.* Annals of Operations Research, 1992. **40**(1): p. 17-32.
36. Berman, O. and D. Krass, *11 Facility Location Problems with Stochastic Demands and Congestion.* Facility location: Applications and theory, 2002: p. 329.
37. Rojeski, P. and C. ReVelle, *Central facilities location under an investment constraint.* Geographical Analysis, 1970. **2**(4): p. 343-360.
38. Gerrard, R.A. and R.L. Church, *Closest assignment constraints and location models: properties and structure.* Location Science, 1996. **4**(4): p. 251-270.
39. Kleinrock, L., *Queueing Systems. Volume 1: Theory.* 1975.
40. Cordeau, J.F., et al., *A guide to vehicle routing heuristics.* Journal of the Operational Research Society, 2002: p. 512-522.
41. Legato, P. and R.M. Mazza, *Berth planning and resources optimisation at a container terminal via discrete event simulation.* European Journal of Operational Research, 2001. **133**(3): p. 537-547.
42. Verter, V. and S.D. Lapierre, *Location of preventive health care facilities.* Annals of Operations Research, 2002. **110**(1): p. 123-132.
43. Goldberg, J.B., *Operations research models for the deployment of emergency services vehicles.* EMS Management Journal, 2004. **1**(1): p. 20-39.
44. Zaki, A.S., H.K. Cheng, and B.R. Parker, *A simulation model for the analysis and management of an emergency service system.* Socio-Economic Planning Sciences, 1997. **31**(3): p. 173-189.
45. Jung, J.Y., et al., *A simulation based optimization approach to supply chain management under demand uncertainty.* Computers & Chemical Engineering, 2004. **28**(10): p. 2087-2106.
46. Pichitlamken, J., B.L. Nelson, and L.J. Hong, *A sequential procedure for neighborhood selection-of-the-best in optimization via simulation.* European Journal of Operational Research, 2006. **173**(1): p. 283-298.
47. Tyni, T. and J. Ylinen, *Evolutionary bi-objective optimisation in the elevator car routing problem.* European Journal of Operational Research, 2006. **169**(3): p. 960-977.
48. Schwartz, J.D., W.L. Wang, and D.E. Rivera, *Simulation-based optimization of process control policies for inventory management in supply chains.* Automatica, 2006. **42**(8): p. 1311-1320.
49. Litvak, N., et al., *Managing the overflow of intensive care patients.* European Journal of Operational Research, 2008. **185**(3): p. 998-1010.
50. Yeh, J.Y. and W.S. Lin, *Using simulation technique and genetic algorithm to improve the quality care of a hospital emergency department.* Expert Systems with Applications, 2007. **32**(4): p. 1073-1083.
51. Rayward-Smith, V.J., et al., *Modern heuristic search methods*1996: Wiley New York.
52. Teitz, M.B. and P. Bart, *Heuristic methods for estimating the generalized vertex median of a weighted graph.* Operations Research, 1968. **16**(5): p. 955-961.
53. Feo, T.A. and M.G.C. Resende, *Greedy randomized adaptive search procedures.* Journal of global optimization, 1995. **6**(2): p. 109-133.
54. Festa, P. and M.G.C. Resende, *An annotated bibliography of GRASP.* Operations Research Letters, 2004. **8**: p. 67-71.
55. Glover, F., *Tabu search—part I.* ORSA Journal on computing, 1989. **1**(3): p. 190-206.
56. Kohlberg, E., *Equilibrium store locations when consumers minimize travel time plus waiting time.* Economics Letters, 1983. **11**(3): p. 211-216.

57. Kariv, O. and S.L. Hakimi, *An algorithmic approach to network location problems. II: The p-medians.* SIAM Journal on Applied Mathematics, 1979. **37**(3): p. 539-560.
58. Sridharan, R., *The capacitated plant location problem.* European Journal of Operational Research, 1995. **87**(2): p. 203-213.

Anexos: Código C++

Anexo 1: Problema de Localização de Cobertura Máxima

```
#pragma once
////////////////////////////GRASP 1opt//////////////////////////////

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <time.h>
#include <math.h>
#include <stdio.h>

#include "baVector.h"
#include "baMatrix.h"
#include "ourtools.h"
#include "cplex.h"


CPXENVptr env;
CPXLPptr lp;

const int N=55;
const int NP=10;
int ND=55;
const double alpha=0.02;
const double miu=1000;
const double wlim=0.0020;
const int ldist=10;

using namespace std;

//////////////////////define functions//////////////////////////////

     void get_datadist (Matrix<float>& dist,int& size4,const int& ND)
     {
           ifstream dist_in;
           int row;
           int col;
           dist_in.open("C:\\code\\M_m_1(cplex Regret)\\distance.dat");
           Assert(!dist_in.fail(), "Can´t open dist.dat");
           dist_in>>size4;
           dist.resize(size4,ND);
           for (row=0; row<dist.numrows(); row++)
                 for (col=0; col<dist.numcols(); col++)
                 dist_in>> dist[row][col];

     }

     void print_datadist (const Matrix<float>& dist)
     {
           int row;
           int col;
           vprn<<"Distances:";
```

```
	vprn<<endl;

	for (row=0; row<dist.numrows(); row++)
		{
		for (col=0; col<dist.numcols(); col++)
			vprn<<setw(6)<<dist[row][col]<<' ';
			vprn<<endl;
		}
}

struct Dj_str
{
	int node;
	float f;
	int pop;
};


void get_datastruc (Vector<Dj_str>& Dj,int& size5)
{
	ifstream ins4;
	ins4.open("C:\\code\\M_m_1(cplex Regret)\\structure.dat");
	Assert(!ins4.fail(), "Can't open struct1.dat");
	size5=0;
	while(ins4>>ws && !ins4.eof())
	{
		ins4>>Dj[size5].node>>Dj[size5].f>>Dj[size5].pop;
		size5++;
	}

	ins4.close();

}

void print_datastruc(const Vector<Dj_str>& Dj, int& size5)
{
	vprn<<"data structure";
	vprn<<endl;
	int i;
	for(i=0;i<size5;i++)
	{
	vprn<<setw(6)<<Dj[i].node<<setw(6)<<Dj[i].f<<setw(6)<<Dj[i].pop;
	vprn<<endl;
	}
}


void swap(float& s, float& t)
{
	float temp;
	temp=s;
	s=t;
	t=temp;
}

void swapint(int& s, int& t)
{
	int temp;
	temp=s;
	s=t;
	t=temp;
```

```
}


void sort_vector( Vector<float>& V,int N)
{
      int ctf, index_min;
      for(ctf=0;ctf<N-1;ctf++)
      {
            index_min=ctf;
            for(int i=ctf+1;i<N;i++)
                  if (V[i]<V[index_min])
                        index_min=i;
            swap (V[ctf],V[index_min]);
      }
}


void alloc_in (Vector<int>& alloc,const int& ND)
{
      int row;
      for (row=0; row<ND; row++)
            alloc[row]=0;
}


void sort_struc( Vector<Dj_str>& Dj,Matrix<float>& dist,Vector<int>&
sumalloc,int& column,const int N)
{
      int ctf;
      int index_min;

      for(ctf=0;ctf<(N-1);ctf++)
      {
            index_min=ctf;


            for(int row=ctf+1;row<N;row++)

                  if (dist[row][column]<dist[index_min][column])
                        index_min=row;


            swap (dist[ctf][column],dist[index_min][column]);
            swap (Dj[ctf].f,Dj[index_min].f);
            swapint (Dj[ctf].node,Dj[index_min].node);
            swapint (Dj[ctf].pop,Dj[index_min].pop);
            swapint (sumalloc[ctf],sumalloc[index_min]);
      }
}


void unsort( Vector<Dj_str>& Dj,Vector<int>& alloc,Matrix<float>& dist,int&
column,const int N)
{
      int ctf;
      int swaps;
      for(ctf=N-1;ctf>0;ctf--)

      {
            swaps=0;
            for(int row=0;row<ctf;row++)
```

```
                        if (Dj[row].node>Dj[row+1].node)
                        {
                  swapint (Dj[row].node,Dj[row+1].node);
                  swapint (Dj[row].pop,Dj[row+1].pop);
                  swap (Dj[row].f,Dj[row+1].f);
                  swapint (alloc[row],alloc[row+1]);
                  swap(dist[row][column],dist[row+1][column]);
                  swaps++;
                        }
                  if(swaps==0) return;

            }

      }


      void freq_j(Vector<Dj_str>& Dj,Vector<int>& alloc,Matrix<float>& dist,int&
column,int N,Vector<float>& inc, const int& ldist)
      {

            double w;
            double inc1=0;
            inc[column]=0;
            for (int row=0; row<N;row++)
            {
                        inc1= inc1+Dj[row].f;
                        w= (inc1/miu)/(miu*(1-inc1/miu));
                        if (w<=wlim && dist[row][column]<=ldist && alloc[row]==0)
                        {
                              inc[column]=inc[column]+Dj[row].f;
                        }


            }
      }

      void alloc_j(Vector<Dj_str>& Dj,Vector<int>& alloc,Matrix<float>&
dist,Vector<int>& LocP,int N,int& index, int ldist)
      {
            float inc;
            double w;

            inc=0;

            for(int row=0; row<N; row++)
            {
                        inc= inc+Dj[row].f;
                        w=(inc/miu)/(miu*(1-inc/miu));



                        if (w<=wlim && dist[row][index]<=ldist && alloc[row]==0)
                        {

                                    alloc[row]=index+1;
                        }

            }
      }
```

```
void print_alloc(Vector<int>& alloc,int N)
{
	int row;
	vprn<<endl;
	for (row=0; row<N; row++)
	{
		vprn<< alloc[row]<<' ';
	}

}

void print_inc(Vector<float>& inc,int N)
{
	int row;
	vprn<<endl;
	for (row=0; row<N; row++)
	{
		vprn<< inc[row]<<' ';
	}

}


void max_vector( Vector<float>& V,int N,int& index_max)
{
		index_max=0;
		for(int i=0;i<N;i++)

			if (V[i]>V[index_max])
				index_max=i;

}


void loc_in (Vector<int>& loc,const int N)
{
	int column;
	for (column=0; column<N; column++)
		loc[column]=0;

}

void print_loc (Vector<int>& loc,const int N)
{
	int column;
	vprn<<"loc";
	vprn<<endl;

	for (column=0; column<N; column++)
		vprn<<loc[column]<<' ';
			vprn<<endl;

}

int obj(Vector<Dj_str>& Dj,Vector<int>& alloc,int N)
{
	int row;
	int sum;
	sum=0;
```

```
	for(row=0;row<N;row++)
	{
		if(alloc[row]!=0)
			sum=sum+Dj[row].pop;
	}
	return sum;
}


int sumloc(Vector<int>& loc,const int N)
{
	int column;
	int sum=0;
	for (column=0;column<N;column++)
		sum+=loc[column];
	return sum;
}

void sum_alloc(Matrix<int>& alloc,const int N, Vector<int>& sumalloc)
{
	int column;
	int row;
	for (row=0;row<N;row++)
	{

		sumalloc[row]=0;
		for (column=0;column<N;column++)
		sumalloc[row]+=alloc[row][column];
	}
}


void copy_m(Matrix<int>& alloc1a,Matrix<int>& alloc1b, int N)
{
int row;
int column;
	for (row=0; row<N; row++)
	{
		for (column=0; column<N; column++)
		alloc1b[row][column]=alloc1a[row][column];

	}
}

void copy_v(Vector<int>& loc,Vector<int>& locfin,const int& N)
{
int column;
	for (column=0; column<N; column++)
		locfin[column]=loc[column];

}

void copy_v2(Vector<double>& loc,Vector<double>& locfin,const int& N)
{
int column;
	for (column=0; column<N; column++)
		locfin[column]=loc[column];

}
void inc_in (Vector<float>& incp,const int N)
{
```

```
            int column;
            for (column=0; column<N; column++)
                  incp[column]=0;

      }


      int min(Matrix<float>& dist, const int NP,Vector<int>& LocP, int& i)
      {
            int index_min=0;
            for( int column=0; column<NP; column++)
            {
                  if(LocP[column]!=0&&LocP[index_min]!=0)
                  if(dist[i][LocP[column]-1]<dist[i][LocP[index_min]-1])
                        index_min=column;
            }
            return index_min;
      }

      double exp(double x, int n)

{

      double e=1;

for(int i=0;i<n;i++)

      e=e*x;

return e;
}


int factorial (int num)
{
 int result=1;
 for (int i=1; i<=num; ++i)
    result=result*=i;
 return result;
}

////////////////////////////////////1-opt/////////////////////////////////////

      void main()
{

ofstream vprn("C:\\code\\M_m_1(cplex Regret)\\maxcov.dat");

////////////////////////////////////////////////////////////////////////////////

Matrix<float> dist(N,N);
Matrix<int> allocmxt(N,N);
Vector<Dj_str> Dj(N);
Vector<int> alloc(N);
Vector<int> allocfin(N);
Vector<int> allocbest(N);
Vector<int> oldalloc (N);
Vector<float> inc(N);
Vector<float> sum(N);
Vector<double> w(NP);
Vector<double> Q(NP);
```

```
Vector<double> wfin(NP);
Vector<double> Qfin(NP);
Vector<double> wbest(NP);
Vector<double> Qbest(NP);
Vector<double> oldw(NP);
Vector<double> oldQ(NP);
Vector<double> waitingtime(NP);
Vector<double> elementsqueue(NP);
Vector<int> LocP(NP);
Vector<int> LocPfin(NP);
Vector<int> LocPbest(NP);
Vector<float> gammaj(N);
Vector<int> locations(NP);
Vector<int> loc(N);
Vector<double> loc1(N);
Vector<int> allocations(N);
int const ns=10;
Matrix<float> popscenario(N,ns);
Matrix<float> fscenario(N,ns);
Matrix<double> regret1(ns,ns+NP+1);
Matrix<double> regret2(ns,ns+NP+1);
Matrix<double> regret3(ns,ns+NP+1+N);
Vector<double> Vregret(ns);
double regretbest;
double I;
double rho;
double P;

int numr;
//int numl;
int index_max;
int index_min;
int objP;
int oldobjP;
int objective1;
int much;
int jj;
int i;
int ii;
int r;
int j;
int index;
double gamma;
clock_t start,finish;
double duration;
//int column;
int ObjCplex;
int objbest;
Vector<int> LocCplex(NP);
Vector<int> AllocCplex(ND);


double alpha=0.7;

srand(time(0));


int M=1000000000;


///////read data distance//////////
```

```
get_datadist(dist,numr,ND);


     start=clock();


////////////////////////////////////////////////////
//generates the population in the range (800,1800)/////
////////////////////////////////////////////////////

for (int k=0;k<ns;k++)
{

for (i=0;i<N;i++)
{

popscenario[i][k]=(int)(10000*rand()/(RAND_MAX+1))+800;

fscenario[i][k]=popscenario[i][k]*0.01;
}
}

int nscount=0;

while (nscount<ns)
     {
           for (i=0;i<N;i++)
           {
                 Dj[i].node=i+1;
                 Dj[i].pop=popscenario[i][nscount];
                 Dj[i].f=fscenario[i][nscount];
     }


//////////////////////////////////Cplex ////////////////////////////////////

int status;
env= CPXopenCPLEX(&status);
lp = CPXcreateprob(env,&status,"MyproB");

int rcnt = 2*N*N+2*N+1;
double *rhs = new double [rcnt];
char *sense= new char [rcnt];

rhs[0]=NP;
sense[0]='E';

for (i=1;i<N*N+1;i++) {
      rhs[i]=0;
      sense[i]='L';
}

for (i=N*N+1;i<N*N+N+1;i++) {
      rhs[i]=1;
      sense[i]='L';
}

for (i=N*N+N+1;i<N*N+2*N+1;i++) {
      rhs[i]=4000;
      sense[i]='L';
```

```
}

for (i=N*N+2*N+1;i<rcnt;i++) {
      rhs[i]=0;
      sense[i]='E';
}

status = CPXnewrows (env, lp, rcnt, rhs, sense, NULL, NULL);

delete [] rhs;
delete [] sense;

int ccnt = N+N*N;
int nzcnt=0;
double *obj = new double [ccnt];
int *cmatbeg = new int [ccnt];

int *cmatind = new int [N+5*N*N];
double *cmatval = new double [N+5*N*N];

for (j=0;j<N; j++) {

      cmatbeg[j]=nzcnt;
      cmatval[nzcnt]=1;
      cmatind[nzcnt]=0;
      nzcnt++;

      for (i=0;i<N;i++) {
            cmatval[nzcnt]=-1;
            cmatind[nzcnt]=1+j*N+i;
            nzcnt++;
      }
      obj[j]=0;
}

int k=0;
int sk=1;
for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {

            cmatbeg[N+i*N+j]=nzcnt;

            cmatval[nzcnt]=1;
            cmatind[nzcnt]=1+j*N+i;
            nzcnt++;

            cmatval[nzcnt]=1;
            cmatind[nzcnt]=1+N*N+i;
            nzcnt++;

            cmatval[nzcnt]=Dj[i].f;
            cmatind[nzcnt]=1+N*N+N+j;
            nzcnt++;

            if (dist[i][j]>ldist)
                 {
                 cmatval[nzcnt]=1;
            cmatind[nzcnt]=1+j*N+i;
            nzcnt++;
                 }
```

```
            obj[N+j*N+i]=Dj[j].pop;
}

}

status = CPXaddcols (env, lp, ccnt, nzcnt, obj, cmatbeg, cmatind, cmatval, NULL,
NULL, NULL);

delete [] obj;
delete [] cmatbeg;
delete [] cmatval;
delete [] cmatind;


int *ind = new int [ccnt];
char *ctype = new char [ccnt];

for (int j=0; j<N*N+N; j++) {
      ind[j]=j;
      ctype[j]='B';
}


status = CPXchgctype (env, lp, ccnt, ind, ctype);

delete [] ind;
delete [] ctype;

CPXchgobjsen (env, lp, CPX_MAX);

status = CPXmipopt(env, lp);


int  lpstat = CPXgetstat (env, lp);

double objval;

status = CPXgetobjval (env, lp, &objval);

CPXwriteprob(env,lp,"exp.lp","lp");

double *x = new double [N+N*N];

status = CPXgetx (env, lp, x, 0, N+N*N-1);

for (j = N; j < N+N*N; j++) cout << x[j]<<" ";

CPXsolwrite(env,lp,"solution.sol");

CPXfreeprob(env,&lp);

CPXcloseCPLEX(&env);

for(j=0; j<N; j++)
loc1[j]=x[j];

for(j=0; j<N; j++)
loc[j]=x[j];
```

```
j=0;
for(i=0; i<N; i++)
if(loc1[i]!=0)
{LocP[j]=i+1;
j+=1;}


int k2=ns;
for(int k1=0; k1<N;k1++)
                if(loc1[k1]!=0)
                      {
                      regret3[nscount][k2]=k1+1;
                      k2+=1;
                }


k=N;
     for(int k1=0;k1<N;k1++)
           for(int k2=0;k2<N;k2++)
           {
                 allocmxt[k1][k2]=x[k];
                 k+=1;
           }


for (i=0; i<N; i++)
alloc[i]=0;

for (int i=0; i<N; i++)
                {
                for (int k=0;k<N; k++)
                      if(allocmxt[i][k]==1)
                      alloc[i]=k+1;

                }

k=0;
for(int i=ns+NP+1;i<ns+NP+N+1;i++)
                {
                      regret3[nscount][i]=alloc[k];
                      k+=1;
                }

delete [] x;

regret1[nscount][nscount]=objval;

for (k=0;k<ns;k++)
{
     if(k!=nscount)
     {
     for (i=0;i<N;i++)
           {
                 Dj[i].pop=popscenario[i][k];
                 Dj[i].f=fscenario[i][k];
           }
int sum=0;

           for(int row=0;row<N;row++)
           {
                 if(alloc[row]!=0)
```

```
                        sum=sum+Dj[row].pop;
            }

      regret1[nscount][k]=sum;

//////////////////////////feasibility////////////////////////////////

      inc_in(inc,N);

      for(int j=0;j<NP;j++)
      {
            for(int i=0;i<N;i++)
            {

                        if(alloc[i]==LocP[j])
                        {
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }

            }
      }

      for(j=0;j<NP;j++)
      {
            w[j]=(inc[LocP[j]-1]/miu)/(miu*(1-(inc[LocP[j]-1]/miu)));
            Q[j]=((inc[LocP[j]-1]/miu)*(inc[LocP[j]-1]/miu))/(1-(inc[LocP[j]-
1]/miu));

      }

      for(j=0;j<NP;j++)
      {
            if (w[j]>wlim)
                  {
                        regret1[nscount][k]=0;
                  }
      }

      for(j=0;j<NP;j++)
      {
            if (w[j]<0)
                  {
                        regret1[nscount][k]=0;
                  }
      }

//////////////////////////fim da feasibility///// ////////////////////////

}
}

nscount+=1;

}
      for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                        regret2[k][i]=-regret1[k][k]+regret1[k][i];
                  }

for (int k=0; k<ns; k++)
```

```
                {
                for (int i=0;i<ns; i++)
                {
                      regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                      if(regret3[k][i]<0)
                            regret3[k][i]=-1*regret3[k][i];
                      if (regret1[k][i]==0)
                            regret3[k][i]=-1*regret3[k][i];
                }

                }

for(int k=0;k<ns;k++)
{
                int index_max=0;
                regret3[k][ns+NP]=regret3[k][index_max];
                for(int i=0;i<ns;i++)
                      if (regret3[k][i]>regret3[k][index_max])
                      {
                            index_max=i;
                            regret3[k][ns+NP]=regret3[k][index_max];
                      }
}
           index_min=0;
           for( int column=0; column<ns; column++)
           {

     if(regret3[column][ns+NP]<regret3[index_min][ns+NP]&&regret3[column][ns+NP]
!=0)
                      index_min=column;
           }

           for(int k=0;k<NP;k++)
           LocP[k]=regret3[index_min][ns+k];


           for(int k=0;k<N;k++)
           alloc[k]=regret3[index_min][ns+NP+k+1];

ObjCplex=regret1[index_min][index_min];

     for(int k=0;k<NP;k++)
           LocCplex[k]=LocP[k];


     for(int k=0;k<ND;k++)
           AllocCplex[k]=alloc[k];

//////////////////////////initialize best results//////////////////////////////

     for(int k=0;k<NP;k++)
           LocPbest[k]=LocP[k];

vprn<<endl<<"initial locations   "<<endl;
     for(int k=0;k<NP;k++)
           vprn<<LocPbest[k]<<" ";

     for(int k=0;k<N;k++)
           allocbest[k]=alloc[k];
```

```
     for(int k=0;k<N;k++)

     regretbest=regret3[index_min][ns+NP];

     objbest=regret1[index_min][index_min];

////////// GRASP//////////////////////////////////////////////////

int m;

     int n_iter=10;

     int iter=0;

     objbest=0;

//////////Initialization//////////////////////////////////////////////////

     for (i=0;i<N;i++)
           {
                 Dj[i].pop=popscenario[i][index_min];
                 Dj[i].f=fscenario[i][index_min];
           }

     inc_in(inc,N);

for (int column=0; column<N; column++)
           {

                       sort_struc(Dj,dist,alloc,column,N);
                       freq_j(Dj,alloc,dist,column,N,inc,ldist);
                       unsort(Dj,alloc,dist,column,N);

           }

           max_vector(inc,N,index_max);

           for (int j=0; j<N; j++)
           {
                 gammaj[j]=inc[j];
           }

           gamma=inc[index_max];

while (iter<n_iter)
{

label1:;

     iter+=1;

           index=(int)(ND*rand()/(RAND_MAX+1.0));

int index2;

           index2=(int)(NP*rand()/(RAND_MAX+1.0));

           if (gammaj[index]>=alpha*gamma)
           {

                 int spy2=0;
```

```
                for (int column3=0; column3<NP; column3++)
                      {
                            if (LocP[column3]==index+1)
                            {
                            spy2=1;
                            break;
                            }
                      }

                if(spy2==0)
                            LocP[index2]=index+1;

                else goto label1;

          }
          else goto label1;

//////////////////initial allocations : nearest locations//////////////////////

     for(i=0;i<ND;i++){
     alloc[i]=LocP[0];
     for(j=0;j<NP;j++){
           if(dist[i][LocP[j]-1]<dist[i][alloc[i]-1])
                 alloc[i]=LocP[j];}

     if(dist[i][alloc[i]-1]>ldist)
           alloc[i]=0;

     }

////////////////////////////compute objective////////////////////////////////

inc_in(inc,N);

objP=0;

     for(int j=0;j<NP;j++)
     {
           for(int i=0;i<N;i++)
           {
                 if(dist[i][LocP[j]-1]<=ldist)
                 {
                       if(alloc[i]==LocP[j])
                       {
                       objP=objP+Dj[i].pop;
                       alloc[i]=LocP[j];
                       inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                       }

                 }
           }
     }

     for(j=0;j<NP;j++)
     {
           w[j]=(inc[LocP[j]-1]/miu)/(miu*(1-(inc[LocP[j]-1]/miu)));
           Q[j]=((inc[LocP[j]-1]/miu)*(inc[LocP[j]-1]/miu))/(1-(inc[LocP[j]-
1]/miu));

     }
```

```
     for(j=0;j<NP;j++)
     {
           if (w[j]>wlim)
                 {
                       objP=0;
                 }
     }

     for(j=0;j<NP;j++)
     {
           if (w[j]<0)
                 {
                       objP=0;
                 }
     }

//////////////local search allocations/////////////////////////////////////////

oldobjP=objP;

for(i=0;i<ND;i++)
for(j=0;j<NP;j++)
if(alloc[i]!=LocP[j]&&dist[i][LocP[j]-1]<ldist)
{
     oldalloc[i]=alloc[i];
     alloc[i]=LocP[j];

//////////////////////Compute Objective/////////////////////////////////////

     inc_in(inc,N);
     objP=0;

     for(int j=0;j<NP;j++)
     {
           for(int i=0;i<N;i++)
           {
                 if(dist[i][LocP[j]-1]<=ldist)
                 {
                       if(alloc[i]==LocP[j])
                       {
                       objP=objP+Dj[i].pop;
                       alloc[i]=LocP[j];
                       inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                       }

                 }
           }
     }

     for(j=0;j<NP;j++)
     {
           w[j]=(inc[LocP[j]-1]/miu)/(miu*(1-(inc[LocP[j]-1]/miu)));
           Q[j]=((inc[LocP[j]-1]/miu)*(inc[LocP[j]-1]/miu))/(1-(inc[LocP[j]-
1]/miu));

     }

     for(j=0;j<NP;j++)
     {
           if (w[j]>wlim)
```

```
                {
                        objP=0;
                }

        }

        for(j=0;j<NP;j++)
        {
                if (w[j]<0)
                        {
                        objP=0;
                }

        }

if(objP>oldobjP)
{
        oldobjP=objP;
}
else
alloc[i]=oldalloc[i];

}

//////////////////////compute regret////////////////////////////////////////

regret1[index_min][index_min]=objP;


for (int k=0;k<ns;k++)
{
        if(k!=index_min)
        {

                for (i=0;i<N;i++)
                {
                        Dj[i].pop=popscenario[i][k];
                        Dj[i].f=fscenario[i][k];
                }



                int sum=0;

                for(int row=0;row<N;row++)
                {
                        if(alloc[row]!=0)
                                sum=sum+Dj[row].pop;
                }

        regret1[index_min][k]=sum;

/////////////////feasibility////////////////////////////////////////////////


        inc_in(inc,N);

        for(int j=0;j<NP;j++)
        {
                for(int i=0;i<N;i++)
                {
```

```
                        if(alloc[i]==LocP[j])
                        {
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }
            }
      }

      for(j=0;j<NP;j++)
      {
            w[j]=(inc[LocP[j]-1]/miu)/(miu*(1-(inc[LocP[j]-1]/miu)));
            Q[j]=((inc[LocP[j]-1]/miu)*(inc[LocP[j]-1]/miu))/(1-(inc[LocP[j]-
1]/miu));

      }

      for(j=0;j<NP;j++)
      {
            if (w[j]>wlim)
                  {
                        regret1[index_min][k]=0;
                  }

      }

      for(j=0;j<NP;j++)
      {
            if (w[j]<0)
                  {
                        regret1[index_min][k]=0;
                  }

      }
      }
}

//////////regret 2/////////////////////////////////////////////////////////////

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                        regret2[k][i]=-regret1[k][k]+regret1[k][i];
                  }

//////////regret 3/////////////////////////////////////////////////////////////

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                  {
                        regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                        if(regret3[k][i]<0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][i]==0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][k]==0)
                              regret3[k][i]=1;

                  }
```

```
                }

int i=0;
for(int k=ns;k<ns+NP;k++)
{
     regret3[index_min][k]=LocP[i];
     i+=1;
}

i=0;
for(int k=ns+NP+1;k<ns+NP+N+1;k++)
{
     regret3[index_min][k]=alloc[i];
     i+=1;
}

for(int k=0;k<ns;k++)
{
                int index_max=0;
                regret3[k][ns+NP]=regret3[k][index_max];
                for(int i=0;i<ns;i++)
                      if (regret3[k][i]>regret3[k][index_max])
                      {
                            index_max=i;
                            regret3[k][ns+NP]=regret3[k][index_max];
                      }
}
           index_min=0;
           for( int column=0; column<ns; column++)
           {

     if(regret3[column][ns+NP]<regret3[index_min][ns+NP]&&regret3[column][ns+NP]
!=0)
                      index_min=column;
           }

           for(int k=0;k<NP;k++)
           LocP[k]=regret3[index_min][ns+k];

           for(int k=0;k<N;k++)
           alloc[k]=regret3[index_min][ns+NP+k+1];

if(regret3[index_min][ns+NP]<regretbest)
{
           regretbest=regret3[index_min][ns+NP];
           for(i=0;i<NP;i++)
           LocPbest[i]=LocP[i];
           for(i=0;i<N;i++)
           allocbest[i]=alloc[i];
}
}

vprn<<endl<<endl<<"FINAL RESULT....."<<endl;

vprn<<endl<<"Regret    "<<endl;

vprn<<regretbest<<" ";

vprn<<endl<<"Locations   "<<endl;

     for(int k=0;k<NP;k++)
```

```
            vprn<<LocPbest[k]<<" ";

objbest=0;

            for(int i=0;i<N;i++)
            {
                  if(alloc[i]!=0)
                        {
                        objbest=objbest+Dj[i].pop;
                        }
                  }
finish=clock();

duration= (double) (finish-start)/CLOCKS_PER_SEC;

vprn<<endl;
exit(0);

}
```

Anexo 2: Problema de Localização de Captura Máxima

```
#pragma once
////////////////////////////GRASP -
1opt//////////////////////////////
////////////////////////////////////////////////////////////////////////
//

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <time.h>
#include <math.h>
#include <stdio.h>
#include "baVector.h"
#include "baMatrix.h"
#include "ourtools.h"
#include "cplex.h"

CPXENVptr env;
CPXLPptr lp;

const int N=50;
const int NP=20;
int ND=50;
const double alpha=0.02;
const double miu=1000;
const double wlim=0.0020;
const int ldist=1000000;

using namespace std;

//////////////////////definefunctions////////////////////////////////

      void get_datadist (Matrix<float>& dist,int& size4,const int& ND)
      {
            ifstream dist_in;
            int row;
            int col;
            dist_in.open("C:\\code\\MaxCap\\distance.dat");
            Assert(!dist_in.fail(), "Can´t open dist.dat");
            dist_in>>size4;
            dist.resize(size4,ND);
            for (row=0; row<dist.numrows(); row++)
                  for (col=0; col<dist.numcols(); col++)
                  dist_in>> dist[row][col];

      }

      void print_datadist (const Matrix<float>& dist)
      {
            int row;
            int col;
            vprn<<"Distances:";
            vprn<<endl;

            for (row=0; row<dist.numrows(); row++)
```

```
            {
            for (col=0; col<dist.numcols(); col++)
                  vprn<<setw(6)<<dist[row][col]<<' ';
                  vprn<<endl;
            }
}

struct Dj_str
{
      int node;
      float f;
      int pop;
};

void get_datastruc (Vector<Dj_str>& Dj,int& size5)
{
      ifstream ins4;
      ins4.open("C:\\code\\MaxCap\\structure.dat");
      Assert(!ins4.fail(), "Can't open struct1.dat");
      size5=0;
      while(ins4>>ws && !ins4.eof())
      {
            ins4>>Dj[size5].node>>Dj[size5].f>>Dj[size5].pop;
            size5++;
      }

      ins4.close();

}

void print_datastruc(const Vector<Dj_str>& Dj, int& size5)
{
      vprn<<"data structure";
      vprn<<endl;
      int i;
      for(i=0;i<size5;i++)
      {

vprn<<setw(6)<<Dj[i].node<<setw(6)<<Dj[i].f<<setw(6)<<Dj[i].pop;
      vprn<<endl;
      }
}

void swap(float& s, float& t)
{
      float temp;
      temp=s;
      s=t;
      t=temp;
}

void swapint(int& s, int& t)
{
      int temp;
      temp=s;
      s=t;
      t=temp;
}


void sort_vector( Vector<float>& V,int N)
```

```
	{
		int ctf, index_min;
		for(ctf=0;ctf<N-1;ctf++)
		{
			index_min=ctf;
			for(int i=ctf+1;i<N;i++)
				if (V[i]<V[index_min])
					index_min=i;
			swap (V[ctf],V[index_min]);
		}
	}

	void alloc_in (Vector<int>& alloc,const int& ND)
	{
		int row;
		for (row=0; row<ND; row++)
			alloc[row]=0;
	}

void sort_struc( Vector<Dj_str>& Dj,Matrix<float>& dist,Vector<int>&
sumalloc,int& column,const int N)
	{
		int ctf;
		int index_min;

		for(ctf=0;ctf<(N-1);ctf++)
		{
			index_min=ctf;

			for(int row=ctf+1;row<N;row++)

				if (dist[row][column]<dist[index_min][column])
					index_min=row;

			swap (dist[ctf][column],dist[index_min][column]);
			swap (Dj[ctf].f,Dj[index_min].f);
			swapint (Dj[ctf].node,Dj[index_min].node);
			swapint (Dj[ctf].pop,Dj[index_min].pop);
			swapint (sumalloc[ctf],sumalloc[index_min]);
		}
	}

	void unsort( Vector<Dj_str>& Dj,Vector<int>&
alloc,Matrix<float>& dist,int& column,const int N)
	{
		int ctf;
		int swaps;
		for(ctf=N-1;ctf>0;ctf--)

		{
			swaps=0;
			for(int row=0;row<ctf;row++)

				if (Dj[row].node>Dj[row+1].node)
				{
			swapint (Dj[row].node,Dj[row+1].node);
			swapint (Dj[row].pop,Dj[row+1].pop);
			swap (Dj[row].f,Dj[row+1].f);
			swapint (alloc[row],alloc[row+1]);
			swap(dist[row][column],dist[row+1][column]);
			swaps++;
```

```
                        }
                  if(swaps==0) return;

            }

      }

      void freq_j(Vector<Dj_str>& Dj,Vector<int>& alloc,Matrix<float>&
dist,int& column,int N,Vector<float>& inc, const int& ldist)
      {

            double w;
            double inc1=0;
            inc[column]=0;
            for (int row=0; row<N;row++)
            {
                        inc1= inc1+Dj[row].f;
                        w= (inc1/miu)/(miu*(1-inc1/miu));
                        if (w<=wlim && dist[row][column]<=ldist &&
alloc[row]==0)
                        {
                              inc[column]=inc[column]+Dj[row].f;
                        }
            }
      }

      void alloc_j(Vector<Dj_str>& Dj,Vector<int>&
alloc,Matrix<float>& dist,Vector<int>& LocP,int N,int& index, int
ldist)
      {
            float inc;
            double w;

            inc=0;

            for(int row=0; row<N; row++)
            {
                        inc= inc+Dj[row].f;
                        w=(inc/miu)/(miu*(1-inc/miu));


                        if (w<=wlim && dist[row][index]<=ldist &&
alloc[row]==0)
                        {

                                    alloc[row]=index+1;
                        }

            }
      }

      void print_alloc(Vector<int>& alloc,int N)
      {
            int row;
            vprn<<endl;
            for (row=0; row<N; row++)
            {
                  vprn<< alloc[row]<<' ';
            }

      }
```

```
void print_inc(Vector<float>& inc,int N)
{
      int row;
      vprn<<endl;
      for (row=0; row<N; row++)
      {
            vprn<< inc[row]<<' ';
      }

}

void max_vector( Vector<float>& V,int N,int& index_max)
{
            index_max=0;
            for(int i=0;i<N;i++)

                  if (V[i]>V[index_max])
                        index_max=i;
}

void loc_in (Vector<int>& loc,const int N)
{
      int column;
      for (column=0; column<N; column++)
            loc[column]=0;
}

void print_loc (Vector<int>& loc,const int N)
{
      int column;
      vprn<<"loc";
      vprn<<endl;

      for (column=0; column<N; column++)
            vprn<<loc[column]<<' ';
                  vprn<<endl;
}

int obj(Vector<Dj_str>& Dj,Vector<int>& alloc,int N)
{
      int row;
      int sum;
      sum=0;
      for(row=0;row<N;row++)
      {
            if(alloc[row]!=0)
                  sum=sum+Dj[row].pop;
      }
      return sum;
}

int sumloc(Vector<int>& loc,const int N)
{
      int column;
      int sum=0;
      for (column=0;column<N;column++)
            sum+=loc[column];
      return sum;
}
```

```
	void sum_alloc(Matrix<int>& alloc,const int N, Vector<int>&
sumalloc)
	{
		int column;
		int row;
		for (row=0;row<N;row++)
		{

			sumalloc[row]=0;
			for (column=0;column<N;column++)
			sumalloc[row]+=alloc[row][column];
		}
	}

	void copy_m(Matrix<int>& alloc1a,Matrix<int>& alloc1b, int N)
	{
	int row;
	int column;
		for (row=0; row<N; row++)
		{
			for (column=0; column<N; column++)
			alloc1b[row][column]=alloc1a[row][column];

		}
	}

	void copy_v(Vector<int>& loc,Vector<int>& locfin,const int& N)
	{
	int column;
		for (column=0; column<N; column++)
			locfin[column]=loc[column];

	}

	void copy_v2(Vector<double>& loc,Vector<double>& locfin,const
int& N)
	{
	int column;
		for (column=0; column<N; column++)
			locfin[column]=loc[column];

	}
	void inc_in (Vector<float>& incp,const int N)
	{
		int column;
		for (column=0; column<N; column++)
			incp[column]=0;

	}

	int min(Matrix<float>& dist, const int NP,Vector<int>& LocP,
int& i)
	{
		int index_min=0;
		for( int column=0; column<NP; column++)
		{
			if(LocP[column]!=0&&LocP[index_min]!=0)
			if(dist[i][LocP[column]-1]<dist[i][LocP[index_min]-
1])
				index_min=column;
		}
```

```
            return index_min;
      }

      double exp(double x, int n)

{

      double e=1;

for(int i=0;i<n;i++)

      e=e*x;

return e;
}


int factorial (int num)
{
 int result=1;
 for (int i=1; i<=num; ++i)
    result=result*=i;
 return result;
}

//////////////////////////1-opt/////////////////////////////////////

      void main()
{

ofstream vprn("C:\\code\\MaxCap\\maxcov.dat");

////////////////////////////////////////////////////////////////////

Matrix<float> dist(N,N);
Matrix<int> allocmxt(N,N);
Vector<Dj_str> Dj(N);
Vector<int> alloc(N);
Vector<int> allocfin(N);
Vector<int> allocbest(N);
Vector<int> oldalloc (N);
Vector<float> inc(N);
Vector<float> sum(N);
Vector<double> w(NP);
Vector<double> Q(NP);
Vector<double> wfin(NP);
Vector<double> Qfin(NP);
Vector<double> wbest(NP);
Vector<double> Qbest(NP);
Vector<double> oldw(NP);
Vector<double> oldQ(NP);
Vector<double> waitingtime(NP);
Vector<double> elementsqueue(NP);
Vector<int> LocP(NP);
Vector<int> LocPfin(NP);
Vector<int> LocPbest(NP);
Vector<float> gammaj(N);
Vector<int> locations(NP);
Vector<int> loc(N);
Vector<double> loc1(N);
Vector<int> allocations(N);
```

```
int const ns=10;
Matrix<float> popscenario(N,ns);
Matrix<float> fscenario(N,ns);
Matrix<double> regret1(ns,ns+NP+1);
Matrix<double> regret2(ns,ns+NP+1);
Matrix<double> regret3(ns,ns+NP+1+N);
Vector<double> Vregret(ns);
double regretbest;
double I;
double rho;
double P;

int numr;
int index_max;
int index_min;
int objP;
int oldobjP;
int objective1;
int much;
int jj;
int i;
int ii;
int r;
int j;
int index;
double gamma;
clock_t start,finish;
double duration;
int ObjCplex;
int objbest;
Vector<int> LocCplex(NP);
Vector<int> AllocCplex(ND);
Vector<float> dibi(N);

double alpha=0.7;

srand(time(0));

int M=1000000000;

///////read data distance//////////
get_datadist(dist,numr,ND);

///////read data dibi//////////////

int size7;

            ifstream ins6;
            ins6.open("C:\\code\\MaxCap\\dibi50.dat");
            Assert(!ins6.fail(), "Can't open dibi50.dat");
            size7=0;
            while(ins6>>ws && !ins6.eof())
            {
                  ins6>>dibi[size7];
                  size7++;
            }

            ins6.close();

for (i=0;i<N;i++)
dibi[i]=dibi[i]/4;
```

```
for(int exam=0;exam<40;exam++)
{


//////////////////////////////////////////////////
//generates the population in the range (800,1800)/////
//////////////////////////////////////////////////

for (int k=0;k<ns;k++)
{

for (i=0;i<N;i++)
{

popscenario[i][k]=(int)(10000*rand()/(RAND_MAX+1))+800;

fscenario[i][k]=popscenario[i][k]*0.01;
}
}

int nscount=0;

while (nscount<ns)
      {
            for (i=0;i<N;i++)
            {
                  Dj[i].node=i+1;
                  Dj[i].pop=popscenario[i][nscount];
                  Dj[i].f=fscenario[i][nscount];
            }

start=clock();

////////////////////////////////////////////////////////////////////
/////////////////////////////////Cplex ////////////////////////////////
////////////////////////////////////////////////////////////////////

int status;
env= CPXopenCPLEX(&status);
lp = CPXcreateprob(env,&status,"MyproB");

int rcnt = 3*N+1;
double *rhs = new double [rcnt];
char *sense= new char [rcnt];

// First row is for the number of facilities to be open

rhs[0]=NP;
sense[0]='E';

// Second set of constraints are linking constraints

for (i=1;i<2*N+1;i++) {
      rhs[i]=0;
      sense[i]='L';
}

// Third set of constraints forces the assignment of each customer to
at most one facility
```

```
for (i=2*N+1;i<3*N+1;i++) {
      rhs[i]=1;
      sense[i]='L';
}

status = CPXnewrows (env, lp, rcnt, rhs, sense, NULL, NULL);

delete [] rhs;
delete [] sense;

// Rows are created

// Now, columns with the coefficients in each row

int ccnt = 3*N; int nzcnt=0;
double *obj = new double [ccnt];
int *cmatbeg = new int [ccnt];

int *cmatind = new int [2*N*N+5*N];
double *cmatval = new double [2*N*N+5*N];

// for the y variables

for (j=0;j<N; j++) {

      // First Y block

      cmatbeg[j]=nzcnt;

      cmatval[nzcnt]=1;
      cmatind[nzcnt]=j+1;
      nzcnt++;

        cmatval[nzcnt]=1;
      cmatind[nzcnt]=2*N+j+1;
        nzcnt++;

        obj[j]=Dj[j].f;

}
for (j=N;j<2*N; j++) {

      // Second Y(Z) block

      cmatbeg[j]=nzcnt;

      cmatval[nzcnt]=1;
      cmatind[nzcnt]=j+1;
      nzcnt++;

        cmatval[nzcnt]=1;
      cmatind[nzcnt]=N+j+1;
        nzcnt++;

        obj[j]=(Dj[j-N].f)/2;

}

// for the x variables

      for (j=2*N; j<3*N; j++)
```

```
        {
            cmatbeg[j]=nzcnt;

            cmatval[nzcnt]=1;
            cmatind[nzcnt]=0;
            nzcnt++;
                  obj[j]=0;

                  for (int i=0; i<N; i++){
                                    if (dist[i][j-2*N]<dibi[i])
                              {
                              cmatval[nzcnt]=-1;
                              cmatind[nzcnt]=1+i;
                              nzcnt++;
                              }
                                          if (dist[i][j-2*N]==dibi[i])
                              {
                              cmatval[nzcnt]=-1;
                              cmatind[nzcnt]=N+i+1;
                              nzcnt++;
                              }
                  }
}
status = CPXaddcols (env, lp, ccnt, nzcnt, obj, cmatbeg, cmatind,
cmatval, NULL, NULL, NULL);

delete [] obj;
delete [] cmatbeg;
delete [] cmatval;
delete [] cmatind;

int *ind = new int [ccnt];
char *ctype = new char [ccnt];

for (int j=0; j<3*N; j++) {
      ind[j]=j;
      ctype[j]='B';
}

status = CPXchgctype (env, lp, ccnt, ind, ctype);
delete [] ind;
delete [] ctype;

CPXchgobjsen (env, lp, CPX_MAX);

status = CPXmipopt(env, lp);

int  lpstat = CPXgetstat (env, lp);

double objval;

status = CPXgetobjval (env, lp, &objval);

CPXwriteprob(env,lp,"exp.lp","lp");

double *x = new double [3*N];

status = CPXgetx (env, lp, x, 0, 3*N-1);

for (j = 0; j < 3*N; j++) cout << x[j]<<" ";
```

```
CPXsolwrite(env,lp,"solution.sol");

CPXfreeprob(env,&lp);

CPXcloseCPLEX(&env);

int k=0;
for(j=0; j<N; j++)
{
loc1[j]=x[2*N+k];
k+=1;
}

k=0;
for(j=0; j<N; j++)
{
loc[j]=x[2*N+k];
k+=1;
}

j=0;
for(i=0; i<N; i++)
if(loc1[i]!=0)
{
      LocP[j]=i+1;
      j+=1;
}

int k2=ns;
for(int k1=0; k1<N;k1++)
                  if(loc1[k1]!=0)
                        {
                        regret3[nscount][k2]=k1+1;
                        k2+=1;
                  }

Vector<int> y(N);
Vector<int> z(N);

for(int i=0;i<N;i++)
      y[i]=x[i];

for(int i=0;i<N;i++)
      z[i]=x[i+N];

delete [] x;

regret1[nscount][nscount]=objval;

for (k=0;k<ns;k++)
{
      if(k!=nscount)
      {
      for (i=0;i<N;i++)
            {
                  Dj[i].pop=popscenario[i][k];
                  Dj[i].f=fscenario[i][k];
            }
int sum=0;

            for(int row=0;row<N;row++)
```

```
            {
                  if(y[row]!=0)
                        sum=sum+Dj[row].f;
                  if(z[row]!=0)
                        sum=sum+Dj[row].f/2;
            }

      regret1[nscount][k]=sum;

nscount+=1;
}

      for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                        regret2[k][i]=-regret1[k][k]+regret1[k][i];
                  }

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                  {
                        regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                        if(regret3[k][i]<0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][i]==0)
                              regret3[k][i]=-1*regret3[k][i];
                  }

                  }

for(int k=0;k<ns;k++)
{
                  int index_max=0;
                  regret3[k][ns+NP]=regret3[k][index_max];
                  for(int i=0;i<ns;i++)
                        if (regret3[k][i]>regret3[k][index_max])
                        {
                              index_max=i;
                              regret3[k][ns+NP]=regret3[k][index_max];
                        }
}

            index_min=0;
            for( int column=0; column<ns; column++)
            {

      if(regret3[column][ns+NP]<regret3[index_min][ns+NP]&&regret3[col
umn][ns+NP]!=0)
                        index_min=column;
            }

            for(int k=0;k<NP;k++)
            LocP[k]=regret3[index_min][ns+k];


            for(int k=0;k<N;k++)
            alloc[k]=regret3[index_min][ns+NP+k+1];

ObjCplex=regret1[index_min][index_min];
```

```
      for(int k=0;k<NP;k++)
            LocCplex[k]=LocP[k];

////initialize best results///////

      for(int k=0;k<NP;k++)
            LocPbest[k]=LocP[k];

      for(int k=0;k<NP;k++)
            vprn<<LocPbest[k]<<" ";

      regretbest=regret3[index_min][ns+NP];

      objbest=regret1[index_min][index_min];

//////////////////////////////////////////////////////////////////////
//////////GRASP///////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

int m;

      int n_iter=500;

      int iter=0;

      objbest=0;

//////////////////////////////////////////////////////////////////////
///////////Initialization/////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

      for (i=0;i<N;i++)
            {
                  Dj[i].pop=popscenario[i][index_min];
                  Dj[i].f=fscenario[i][index_min];
            }



//////////////////initial allocations : nearest locations///////////

      for(i=0;i<ND;i++){
      alloc[i]=LocP[0];
      for(j=0;j<NP;j++){
            if(dist[i][LocP[j]-1]<dist[i][alloc[i]-1])
                  alloc[i]=LocP[j];}
      }

//////////////////////////////////////////////////////////////////////

      inc_in(inc,N);


for (int column=0; column<N; column++)
            {

                        sort_struc(Dj,dist,alloc,column,N);
                        freq_j(Dj,alloc,dist,column,N,inc,ldist);
                        unsort(Dj,alloc,dist,column,N);
```

```
            }

            max_vector(inc,N,index_max);

            for (int j=0; j<N; j++)
            {
                  gammaj[j]=inc[j];
            }

            gamma=inc[index_max];

while (iter<n_iter)
{

iter+=1;

label1:;
            index=(int)(ND*rand()/(RAND_MAX+1.0));

int index2;

            index2=(int)(NP*rand()/(RAND_MAX+1.0));

            if (gammaj[index]>=alpha*gamma)
            {
                  int spy2=0;

                  for (int column3=0; column3<NP; column3++)
                        {
                              if (LocP[column3]==index+1)
                              {
                              spy2=1;
                              break;
                              }
                        }

                  if(spy2==0)
                              LocP[index2]=index+1;

                  else goto label1;
            }
            else goto label1;

//////////////////initial allocations : nearest locations///////////

      for(i=0;i<ND;i++){
      alloc[i]=LocP[0];
      for(j=0;j<NP;j++){
            if(dist[i][LocP[j]-1]<dist[i][alloc[i]-1])
                  alloc[i]=LocP[j];}

      if(dist[i][alloc[i]-1]>ldist)
            alloc[i]=0;
      }

//////////////////////////////////////////////////////////////////////
//////////////////compute objective/////////////////////////////////
//////////////////////////////////////////////////////////////////////

inc_in(inc,N);
```

```
objP=0;

	for(int j=0;j<NP;j++)
	{
		for(int i=0;i<N;i++)
		{
			if(dist[i][LocP[j]-1]<dibi[i])
			{
				if(alloc[i]==LocP[j])
				{
				objP=objP+Dj[i].f;
				alloc[i]=LocP[j];
				inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
				}
			}

			if(dist[i][LocP[j]-1]==dibi[i])
			{
				if(alloc[i]==LocP[j])
				{
				objP=objP+Dj[i].f/2;
				alloc[i]=LocP[j];
				inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
				}
			}
		}
	}

//////////////////////////////////////////////////////////////////////
//////////////////////local search allocations//////////////////////////
//////////////////////////////////////////////////////////////////////

oldobjP=objP;

for(i=0;i<ND;i++)
for(j=0;j<NP;j++)
if(alloc[i]!=LocP[j]&&dist[i][LocP[j]-1]<ldist)
{
	oldalloc[i]=alloc[i];
	alloc[i]=LocP[j];
//////////////////////////////////////////////////////////////////
////////Compute Objective/////////////////////////////////////////
//////////////////////////////////////////////////////////////////

	inc_in(inc,N);
	objP=0;

	for(int j=0;j<NP;j++)
	{
		for(int i=0;i<N;i++)
		{
			if(dist[i][LocP[j]-1]<dibi[i])
			{
				if(alloc[i]==LocP[j])
				{
				objP=objP+Dj[i].f;
				alloc[i]=LocP[j];
				inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
				}
			}
```

```
                  if(dist[i][LocP[j]-1]==dibi[i])
                  {
                        if(alloc[i]==LocP[j])
                        {
                        objP=objP+Dj[i].f/2;
                        alloc[i]=LocP[j];
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }
                  }
            }
      }

if(objP>oldobjP)
{
      oldobjP=objP;
}
else
alloc[i]=oldalloc[i];

}

//////////////////////////compute regret////////////////////////////////

regret1[index_min][index_min]=objP;

for (int k=0;k<ns;k++)
{
      if(k!=index_min)
      {

            for (i=0;i<N;i++)
            {
                  Dj[i].pop=popscenario[i][k];
                  Dj[i].f=fscenario[i][k];
            }

            int sum=0;

            for(int row=0;row<N;row++)
            {
                  if(alloc[row]!=0)
                        sum=sum+Dj[row].f;
            }

      regret1[index_min][k]=sum;

////////////////////////////////////////////////
//////feasibility///////////////////////////////
////////////////////////////////////////////////


      inc_in(inc,N);

      for(int j=0;j<NP;j++)
      {
            for(int i=0;i<N;i++)
            {

                        if(alloc[i]==LocP[j])
                        {
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
```

```
                        }
                }
        }
        }
}

//////////regret 2//////////////////////


for (int k=0; k<ns; k++)
                    {
                    for (int i=0;i<ns; i++)
                          regret2[k][i]=-regret1[k][k]+regret1[k][i];
                    }

//////////regret 3//////////////////////////

for (int k=0; k<ns; k++)
                    {
                    for (int i=0;i<ns; i++)
                    {
                          regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                          if(regret3[k][i]<0)
                                regret3[k][i]=-1*regret3[k][i];
                          if (regret1[k][i]==0)
                                regret3[k][i]=-1*regret3[k][i];
                          if (regret1[k][k]==0)
                                regret3[k][i]=1;
                    }
                    }
int i=0;
for(int k=ns;k<ns+NP;k++)
{
      regret3[index_min][k]=LocP[i];
      i+=1;
}

i=0;
for(int k=ns+NP+1;k<ns+NP+N+1;k++)
{
      regret3[index_min][k]=alloc[i];
      i+=1;
}

for(int k=0;k<ns;k++)
{
                    int index_max=0;
                    regret3[k][ns+NP]=regret3[k][index_max];
                    for(int i=0;i<ns;i++)
                          if (regret3[k][i]>regret3[k][index_max])
                          {
                                index_max=i;
                                regret3[k][ns+NP]=regret3[k][index_max];
                          }
}
            index_min=0;
            for( int column=0; column<ns; column++)
            {
```

```
      if(regret3[column][ns+NP]<regret3[index_min][ns+NP]&&regret3[col
umn][ns+NP]!=0)
                        index_min=column;
            }

            for(int k=0;k<NP;k++)
            LocP[k]=regret3[index_min][ns+k];

            for(int k=0;k<N;k++)
            alloc[k]=regret3[index_min][ns+NP+k+1];

if(regret3[index_min][ns+NP]<regretbest)
{
            regretbest=regret3[index_min][ns+NP];
            for(i=0;i<NP;i++)
            LocPbest[i]=LocP[i];
            for(i=0;i<N;i++)
            allocbest[i]=alloc[i];
}
}

vprn<<regretbest<<" ";

      for(int k=0;k<NP;k++)
            vprn<<LocPbest[k]<<" ";

finish=clock();

duration= (double) (finish-start)/CLOCKS_PER_SEC;

vprn<<duration;
vprn<<endl;

}
exit(0);
}
```

Anexo 3: Problema de Localização de Infraestruturas com Capacidade Limitada

```
#pragma once
////////////////////////////GRASP -1 opt//////////////////////////////

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <time.h>
#include <math.h>
#include <stdio.h>

#include "baVector.h"
#include "baMatrix.h"
#include "ourtools.h"
#include "cplex.h"

CPXENVptr env;
CPXLPptr lp;

const int N=50;
int NP;
int ND=50;
int cap=1000;
const double alpha=0.02;
const double miu=1000;
const double wlim=0.0020;
const int ldist=1000000;

using namespace std;

//////////////////////define functions//////////////////////////////

      void get_datadist (Matrix<float>& dist,int& size4,const int& ND)
      {
            ifstream dist_in;
            int row;
            int col;
            dist_in.open("C:\\code\\MaxLoc\\distance.dat");
            Assert(!dist_in.fail(), "Can´t open dist.dat");
            dist_in>>size4;
            dist.resize(size4,ND);
            for (row=0; row<dist.numrows(); row++)
                  for (col=0; col<dist.numcols(); col++)
                  dist_in>> dist[row][col];

      }

      void print_datadist (const Matrix<float>& dist)
      {
            int row;
            int col;
            vprn<<"Distances:";
            vprn<<endl;
```

```
      for (row=0; row<dist.numrows(); row++)
            {
            for (col=0; col<dist.numcols(); col++)
                  vprn<<setw(6)<<dist[row][col]<<' ';
                  vprn<<endl;
            }
}

struct Dj_str
{
      int node;
      float f;
      int pop;
};

void get_datastruc (Vector<Dj_str>& Dj,int& size5)
{
      ifstream ins4;
      ins4.open("C:\\code\\MaxLoc\\structure.dat");
      Assert(!ins4.fail(), "Can't open struct1.dat");
      size5=0;
      while(ins4>>ws && !ins4.eof())
      {
            ins4>>Dj[size5].node>>Dj[size5].f>>Dj[size5].pop;
            size5++;
      }

      ins4.close();

}

void print_datastruc(const Vector<Dj_str>& Dj, int& size5)
{
      vprn<<"data structure";
      vprn<<endl;
      int i;
      for(i=0;i<size5;i++)
      {

vprn<<setw(6)<<Dj[i].node<<setw(6)<<Dj[i].f<<setw(6)<<Dj[i].pop;
      vprn<<endl;
      }
}

void swap(float& s, float& t)
{
      float temp;
      temp=s;
      s=t;
      t=temp;
}

void swapint(int& s, int& t)
{
      int temp;
      temp=s;
      s=t;
      t=temp;
}

void sort_vector( Vector<float>& V,int N)
```

```
	{
		int ctf, index_min;
		for(ctf=0;ctf<N-1;ctf++)
		{
			index_min=ctf;
			for(int i=ctf+1;i<N;i++)
				if (V[i]<V[index_min])
					index_min=i;
			swap (V[ctf],V[index_min]);
		}
	}

	void alloc_in (Vector<int>& alloc,const int& ND)
	{
		int row;
		for (row=0; row<ND; row++)
			alloc[row]=0;
	}

void sort_struc( Vector<Dj_str>& Dj,Matrix<float>& dist,Vector<int>&
sumalloc,int& column,const int N)
	{
		int ctf;
		int index_min;

		for(ctf=0;ctf<(N-1);ctf++)
		{
			index_min=ctf;

			for(int row=ctf+1;row<N;row++)

				if (dist[row][column]<dist[index_min][column])
					index_min=row;

			swap (dist[ctf][column],dist[index_min][column]);
			swap (Dj[ctf].f,Dj[index_min].f);
			swapint (Dj[ctf].node,Dj[index_min].node);
			swapint (Dj[ctf].pop,Dj[index_min].pop);
			swapint (sumalloc[ctf],sumalloc[index_min]);
		}
	}

	void unsort( Vector<Dj_str>& Dj,Vector<int>&
alloc,Matrix<float>& dist,int& column,const int N)
	{
		int ctf;
		int swaps;
		for(ctf=N-1;ctf>0;ctf--)

		{
			swaps=0;
			for(int row=0;row<ctf;row++)

				if (Dj[row].node>Dj[row+1].node)
				{
			swapint (Dj[row].node,Dj[row+1].node);
			swapint (Dj[row].pop,Dj[row+1].pop);
			swap (Dj[row].f,Dj[row+1].f);
			swapint (alloc[row],alloc[row+1]);
			swap(dist[row][column],dist[row+1][column]);
			swaps++;
```

```
                        }
                  if(swaps==0) return;

            }

      }

      void freq_j(Vector<Dj_str>& Dj,Vector<int>& alloc,Matrix<float>&
dist,int& column,int N,Vector<float>& inc, const int& ldist)
      {
            double w;
            double inc1=0;
            inc[column]=0;
            for (int row=0; row<N;row++)
            {
                        inc1= inc1+Dj[row].f;
                        w= (inc1/miu)/(miu*(1-inc1/miu));
                        if (w<=wlim && dist[row][column]<=ldist &&
alloc[row]==0)
                        {
                              inc[column]=inc[column]+Dj[row].f;
                        }
            }
      }

      void alloc_j(Vector<Dj_str>& Dj,Vector<int>&
alloc,Matrix<float>& dist,Vector<int>& LocP,int N,int& index, int
ldist)
      {
            float inc;
            double w;

            inc=0;

            for(int row=0; row<N; row++)
            {
                        inc= inc+Dj[row].f;
                        w=(inc/miu)/(miu*(1-inc/miu));


                        if (w<=wlim && dist[row][index]<=ldist &&
alloc[row]==0)
                        {

                                    alloc[row]=index+1;
                        }
            }
      }

      void print_alloc(Vector<int>& alloc,int N)
      {
            int row;
            vprn<<endl;
            for (row=0; row<N; row++)
            {
                  vprn<< alloc[row]<<' ';
            }
      }

      void print_inc(Vector<float>& inc,int N)
      {
```

```
	int row;
	vprn<<endl;
	for (row=0; row<N; row++)
	{
		vprn<< inc[row]<<' ';
	}
}

void max_vector( Vector<float>& V,int N,int& index_max)
{
		index_max=0;
		for(int i=0;i<N;i++)

			if (V[i]>V[index_max])
				index_max=i;
}

void loc_in (Vector<int>& loc,const int N)
{
	int column;
	for (column=0; column<N; column++)
		loc[column]=0;
}

void print_loc (Vector<int>& loc,const int N)
{
	int column;
	vprn<<"loc";
	vprn<<endl;

	for (column=0; column<N; column++)
		vprn<<loc[column]<<' ';
			vprn<<endl;

}

int obj(Vector<Dj_str>& Dj,Vector<int>& alloc,int N)
{
	int row;
	int sum;
	sum=0;
	for(row=0;row<N;row++)
	{
		if(alloc[row]!=0)
			sum=sum+Dj[row].pop;
	}
	return sum;
}

int sumloc(Vector<int>& loc,const int N)
{
	int column;
	int sum=0;
	for (column=0;column<N;column++)
		sum+=loc[column];
	return sum;
}

	void sum_alloc(Matrix<int>& alloc,const int N, Vector<int>&
sumalloc)
	{
```

```
            int column;
            int row;
            for (row=0;row<N;row++)
            {

                  sumalloc[row]=0;
                  for (column=0;column<N;column++)
                  sumalloc[row]+=alloc[row][column];
            }
      }

      void copy_m(Matrix<int>& alloc1a,Matrix<int>& alloc1b, int N)
      {
      int row;
      int column;
            for (row=0; row<N; row++)
            {
                  for (column=0; column<N; column++)
                  alloc1b[row][column]=alloc1a[row][column];

            }
      }

      void copy_v(Vector<int>& loc,Vector<int>& locfin,const int& N)
      {
      int column;
            for (column=0; column<N; column++)
                  locfin[column]=loc[column];

      }

      void copy_v2(Vector<double>& loc,Vector<double>& locfin,const
int& N)
      {
      int column;
            for (column=0; column<N; column++)
                  locfin[column]=loc[column];

      }
      void inc_in (Vector<float>& incp,const int N)
      {
            int column;
            for (column=0; column<N; column++)
                  incp[column]=0;

      }

      int min(Matrix<float>& dist, const int NP,Vector<int>& LocP,
int& i)
      {
            int index_min=0;
            for( int column=0; column<NP; column++)
            {
                  if(LocP[column]!=0&&LocP[index_min]!=0)
                  if(dist[i][LocP[column]-1]<dist[i][LocP[index_min]-
1])
                        index_min=column;
            }
            return index_min;
      }
```

```
      double exp(double x, int n)

{
      double e=1;

for(int i=0;i<n;i++)

      e=e*x;

return e;
}

int factorial (int num)
{
 int result=1;
 for (int i=1; i<=num; ++i)
    result=result*=i;
 return result;
}

////////////////////////////////////1-opt//////////////////////////

      void main()
{

ofstream vprn("C:\\code\\MaxLoc\\maxcov.dat");

//////////////////////////////////////////////////////////////////////

Matrix<float> dist(N,N);
Matrix<int> allocmxt(N,N);
Vector<Dj_str> Dj(N);
Vector<int> alloc(N);
Vector<int> allocfin(N);
Vector<int> allocbest(N);
Vector<int> oldalloc (N);
Vector<float> inc(N);
Vector<float> sum(N);
Vector<double> w(NP);
Vector<double> Q(NP);
Vector<double> wfin(NP);
Vector<double> Qfin(NP);
Vector<double> wbest(NP);
Vector<double> Qbest(NP);
Vector<double> oldw(NP);
Vector<double> oldQ(NP);
Vector<double> waitingtime(NP);
Vector<double> elementsqueue(NP);
Vector<int> LocP(NP);
Vector<int> LocPfin(NP);
Vector<int> LocPbest(NP);
Vector<float> gammaj(N);
Vector<int> locations(NP);
Vector<int> loc(N);
Vector<double> loc1(N);
Vector<int> allocations(N);
Vector<float> fi(N);
int const ns=10;
Matrix<float> popscenario(N,ns);
Matrix<float> fscenario(N,ns);
Matrix<double> regret1(ns,ns+NP+1);
```

```
Matrix<double> regret2(ns,ns+NP+1);
Matrix<double> regret3(ns,ns+2*N+2);
Vector<double> Vregret(ns);
Matrix<float> fiscenario(N,ns);
double regretbest;
double I;
double rho;
double P;

int numr;
int index_max;
int index_min;
int objP;
int oldobjP;
int objective1;
int much;
int jj;
int i;
int ii;
int r;
int j;
int index;
double gamma;
clock_t start,finish;
double duration;
int ObjCplex;
int objbest;
Vector<int> LocCplex(NP);
Vector<int> AllocCplex(ND);

double alpha=0.7;

srand(time(0));


int M=1000000000;


///////read data distance//////////
get_datadist(dist,numr,ND);


for(int exam=0;exam<50;exam++)
{
      start=clock();

//////////////////////////////////////////////////////
//generates the population in the range (800,1800)/////
//////////////////////////////////////////////////////

for (int k=0;k<ns;k++)
{

for (i=0;i<N;i++)
{

popscenario[i][k]=(int)(10000*rand()/(RAND_MAX+1))+800;

fscenario[i][k]=popscenario[i][k]*0.01;

fiscenario[i][k]=(float)(500*rand()/(RAND_MAX+1)+100);
```

```
}
}

int nscount=0;

while (nscount<ns)
      {
            for (i=0;i<N;i++)
            {
                  Dj[i].node=i+1;
                  Dj[i].pop=popscenario[i][nscount];
                  Dj[i].f=fscenario[i][nscount];
                  fi[i]=fiscenario[i][nscount];
            }

////////////////////////////////Cplex ////////////////////////////////

cout<<endl<<endl<<"nscount  "<<nscount<<endl<<endl;

// Open cplex environment
int status;
env= CPXopenCPLEX(&status);


lp = CPXcreateprob(env,&status,"MyproB");

int rcnt = 2*N+N*N;
double *rhs = new double [rcnt];
char *sense= new char [rcnt];

for (i=0;i<N*N;i++) {
      rhs[i]=0;
      sense[i]='L';
}

for (i=N*N;i<N*N+N;i++) {
      rhs[i]=1;
      sense[i]='E';
}

for (i=N*N+N;i<N*N+2*N;i++) {
      rhs[i]=cap;
      sense[i]='L';
}

status = CPXnewrows (env, lp, rcnt, rhs, sense, NULL, NULL);

delete [] rhs;
delete [] sense;

int ccnt = N+N*N;
int nzcnt=0;
double *obj = new double [ccnt];
int *cmatbeg = new int [ccnt];

int *cmatind = new int [4*N*N];
double *cmatval = new double [4*N*N];

for (j=0;j<N; j++) {
```

```
      cmatbeg[j]=nzcnt;

      for (i=0;i<N;i++) {
            cmatval[nzcnt]=-1;
            cmatind[nzcnt]=j*N+i;
            nzcnt++;
      obj[i]=fi[i];
        }

}

int k=0;
int sk=1;

for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {

            cmatbeg[N+i*N+j]=nzcnt;

            cmatval[nzcnt]=1;
            cmatind[nzcnt]=j*N+i;
            nzcnt++;

            cmatval[nzcnt]=1;
            cmatind[nzcnt]=N*N+i;
            nzcnt++;

            cmatval[nzcnt]=Dj[i].f;
            cmatind[nzcnt]=N*N+N+j;
            nzcnt++;

            obj[N+j*N+i]=dist[i][j];
}
}

status = CPXaddcols (env, lp, ccnt, nzcnt, obj, cmatbeg, cmatind,
cmatval, NULL, NULL, NULL);

delete [] obj;
delete [] cmatbeg;
delete [] cmatval;
delete [] cmatind;

int *ind = new int [ccnt];
char *ctype = new char [ccnt];

for (int j=0; j<N*N+N; j++) {
      ind[j]=j;
      ctype[j]='B';
}

status = CPXchgctype (env, lp, ccnt, ind, ctype);

delete [] ind;
delete [] ctype;

CPXchgobjsen (env, lp, CPX_MIN);

status = CPXmipopt(env, lp);
```

```
cout<< endl << "Cplex status " << status;

int  lpstat = CPXgetstat (env, lp);

double objval;

status = CPXgetobjval (env, lp, &objval);

CPXwriteprob(env,lp,"exp.lp","lp");

double *x = new double [N+N*N];

status = CPXgetx (env, lp, x, 0, N+N*N-1);

CPXsolwrite(env,lp,"solution.sol");

CPXfreeprob(env,&lp);

CPXcloseCPLEX(&env);

for(j=0; j<N; j++)
loc1[j]=x[j];

for(j=0; j<N; j++)
loc[j]=x[j];

NP=0;

for(i=0; i<N; i++)
if(loc1[i]!=0)
NP+=1;

Vector<int> LocP(NP);

j=0;
for(i=0; i<N; i++)
if(loc1[i]!=0)
{LocP[j]=i+1;
j+=1;}

regret3[nscount][ns]=NP;

int k2=ns+1;
for(int k1=0; k1<N;k1++)
                  if(loc1[k1]!=0)
                        {
                        regret3[nscount][k2]=k1+1;
                        k2+=1;
                  }
k=N;
      for(int k1=0;k1<N;k1++)
            for(int k2=0;k2<N;k2++)
            {
                  allocmxt[k1][k2]=x[k];
                  k+=1;
            }

for (i=0; i<N; i++)
alloc[i]=0;

for (int i=0; i<N; i++)
```

```
                  {
                  for (int k=0;k<N; k++)
                        if(allocmxt[i][k]==1)
                        alloc[i]=k+1;
                  }
k=0;
for(int i=ns+N+2;i<ns+N+N+2;i++)
                  {
                        regret3[nscount][i]=alloc[k];
                        k+=1;
                  }
delete [] x;

regret1[nscount][nscount]=objval;

for (k=0;k<ns;k++)
{
      if(k!=nscount)
      {

      int sum=0;

      for(j=0; j<NP; j++)
      sum+=fiscenario[LocP[j]-1][k];

      for(int j=0;j<NP;j++)
      {
            for(int i=0;i<N;i++)
            {
                  if(alloc[i]==LocP[j])
                        {
                        sum=sum+dist[i][LocP[j]-1];
                        }
            }
      }

regret1[nscount][k]=sum;


////////////////////////////////////////////
//////feasibility///////////////////////////
////////////////////////////////////////////


      inc_in(inc,N);

      for(int j=0;j<NP;j++)
      {
            for(int i=0;i<N;i++)
            {
                        if(alloc[i]==LocP[j])
                        {
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }
            }
      }

/////////fim da feasibility//////////////////////////
}
}
nscount+=1;
```

```
}
      for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                        regret2[k][i]=-regret1[k][k]+regret1[k][i];
                  }

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                  {
                        regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                        if(regret3[k][i]<0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][i]==0)
                              regret3[k][i]=-1*regret3[k][i];
                  }
                  }

for(int k=0;k<ns;k++)
{
                  int index_max=0;
                  regret3[k][ns+N+1]=regret3[k][index_max];
                  for(int i=0;i<ns;i++)
                        if (regret3[k][i]>regret3[k][index_max])
                        {
                              index_max=i;
                              regret3[k][ns+N+1]=regret3[k][index_max];
                        }
}
            index_min=0;
            for( int column=0; column<ns; column++)
            {

      if(regret3[column][ns+N+1]<regret3[index_min][ns+N+1]&&regret3[c
olumn][ns+N+1]!=0)
                        index_min=column;
            }

            NP=regret3[index_min][ns];

            Vector<int> LocP(NP);

            for(int k=0;k<NP;k++)
            LocP[k]=regret3[index_min][ns+k+1];

            for(int k=0;k<N;k++)
            alloc[k]=regret3[index_min][ns+N+k+2];

ObjCplex=regret1[index_min][index_min];

            Vector<int> LocCplex(NP);

      for(int k=0;k<NP;k++)
            LocCplex[k]=LocP[k];

      for(int k=0;k<ND;k++)
            AllocCplex[k]=alloc[k];
```

```
////initialize best results///////

            Vector<int> LocPbest(NP);

      for(int k=0;k<NP;k++)
            LocPbest[k]=LocP[k];

      for(int k=0;k<NP;k++)
            vprn<<LocPbest[k]<<" ";

      for(int k=0;k<N;k++)
            allocbest[k]=alloc[k];

      regretbest=regret3[index_min][ns+NP];

      objbest=regret1[index_min][index_min];

//////////reactive GRASP////////////////////////////////////////////

int m;

      int n_iter=2000;

      int iter=0;

      objbest=0;
//////////Initialization////////////////////////////////////////////

      for (i=0;i<N;i++)
            {
                  Dj[i].pop=popscenario[i][index_min];
                  Dj[i].f=fscenario[i][index_min];
            }

      inc_in(inc,N);

while (iter<n_iter)
{

label1:;

      iter+=1;

            index=(int)(ND*rand()/(RAND_MAX+1.0));

int index2;

            index2=(int)(NP*rand()/(RAND_MAX+1.0));

                  int spy2=0;

                  for (int column3=0; column3<NP; column3++)
                        {
                              if (LocP[column3]==index+1)
                              {
                              spy2=1;
                              break;
                              }
                        }

                  if(spy2==0)
```

```
                              LocP[index2]=index+1;


                  else goto label1;

//////////////////initial allocations : nearest locations///////////

for(i=0;i<ND;i++){
      alloc[i]=LocP[0];
      for(j=0;j<NP;j++){
            if(dist[i][LocP[j]-1]<dist[i][alloc[i]-1])
                  alloc[i]=LocP[j];}

      if(dist[i][alloc[i]-1]>ldist)
            alloc[i]=0;
      }
//////////////compute objective////////////////////////

inc_in(inc,N);

objP=0;

for(j=0; j<NP; j++)
objP+=fi[LocP[j]-1];

      for(int j=0;j<NP;j++)
      {
            for(int i=0;i<N;i++)
            {
                  if(inc[LocP[j]-1]+Dj[i].f<=cap)
                  {
                        if(alloc[i]==LocP[j])
                        {
                        objP=objP+dist[i][LocP[j]-1];
                        alloc[i]=LocP[j];
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }

                  }
            }
      }

//////////////////////////local search allocations/////////

oldobjP=objP;

for(i=0;i<ND;i++)
for(j=0;j<NP;j++)
if(alloc[i]!=LocP[j]&&dist[i][LocP[j]-1]<ldist)
{
      oldalloc[i]=alloc[i];
      alloc[i]=LocP[j];


////////////////////////////////////////////////////////////////
////////Compute Objective///////////////////////////////////////
////////////////////////////////////////////////////////////////

      inc_in(inc,N);
      objP=0;
```

```
for(j=0; j<NP; j++)
objP+=fi[LocP[j]-1];

	for(int j=0;j<NP;j++)
	{
		for(int i=0;i<N;i++)
		{
			if(inc[LocP[j]-1]+Dj[i].f<=cap)
			{
				if(alloc[i]==LocP[j])
				{
				objP=objP+dist[i][LocP[j]-1];
				alloc[i]=LocP[j];
				inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
				}
			}
		}
	}

if(objP<oldobjP)
{
	oldobjP=objP;
}
else
alloc[i]=oldalloc[i];

}

//////////////////////compute regret////////////////////////

regret1[index_min][index_min]=objP;

for (int k=0;k<ns;k++)
{
	if(k!=index_min)
	{

		for (i=0;i<N;i++)
		{
			Dj[i].pop=popscenario[i][k];
			Dj[i].f=fscenario[i][k];
		}

		int sum=0;

		for(int row=0;row<N;row++)
		{
			if(alloc[row]!=0)
				sum=sum+Dj[row].pop;
		}

	regret1[index_min][k]=sum;

////////////////////////////////////////////
//////feasibility///////////////////////////
////////////////////////////////////////////

	inc_in(inc,N);

	for(int j=0;j<NP;j++)
	{
```

```
            for(int i=0;i<N;i++)
            {
                        if(alloc[i]==LocP[j])
                        {
                        inc[LocP[j]-1]=inc[LocP[j]-1]+Dj[i].f;
                        }
            }
      }

//////////regret 2/////////////////////////////////////////

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                        regret2[k][i]=-regret1[k][k]+regret1[k][i];
                  }

//////////regret 3/////////////////////////////////////////////////////

for (int k=0; k<ns; k++)
                  {
                  for (int i=0;i<ns; i++)
                  {
                        regret3[k][i]=(regret1[k][k]-
regret1[k][i])/regret1[k][k];
                        if(regret3[k][i]<0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][i]==0)
                              regret3[k][i]=-1*regret3[k][i];
                        if (regret1[k][k]==0)
                              regret3[k][i]=1;

                  }
                  }

int i=0;
for(int k=ns;k<ns+NP;k++)
{
      regret3[index_min][k]=LocP[i];
      i+=1;
}

i=0;
for(int k=ns+NP+1;k<ns+NP+N+1;k++)
{
      regret3[index_min][k]=alloc[i];
      i+=1;
}

for(int k=0;k<ns;k++)
{
                  int index_max=0;
                  regret3[k][ns+NP]=regret3[k][index_max];
                  for(int i=0;i<ns;i++)
                        if (regret3[k][i]>regret3[k][index_max])
                        {
                              index_max=i;
                              regret3[k][ns+NP]=regret3[k][index_max];
                        }
}
            index_min=0;
```

```
            for( int column=0; column<ns; column++)
            {

      if(regret3[column][ns+NP]<regret3[index_min][ns+NP]&&regret3[col
umn][ns+NP]!=0)
                        index_min=column;
            }

            for(int k=0;k<NP;k++)
            LocP[k]=regret3[index_min][ns+k];


            for(int k=0;k<N;k++)
            alloc[k]=regret3[index_min][ns+NP+k+1];

if(regret3[index_min][ns+NP]<regretbest)
{
            regretbest=regret3[index_min][ns+NP];
            for(i=0;i<NP;i++)
            LocPbest[i]=LocP[i];
            for(i=0;i<N;i++)
            allocbest[i]=alloc[i];
}
}

vprn<<regretbest<<" ";

      for(int k=0;k<NP;k++)
            vprn<<LocPbest[k]<<" ";
objbest=0;
            for(int i=0;i<N;i++)
            {
                  if(alloc[i]!=0)
                        {
                        objbest=objbest+Dj[i].pop;
                        }

                  }
finish=clock();

duration= (double) (finish-start)/CLOCKS_PER_SEC;

vprn<<duration;
vprn<<endl;
}
exit(0);
}
```