Um Algoritmo Guloso para a Configuração e Reconfiguração de *Scatternets* Bluetooth

Wanderley Ravagnani Junior

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Algoritmo Guloso para a Configuração e Reconfiguração de *Scatternets* Bluetooth

Wanderley Ravagnani Junior[1]

7 de Novembro de 2003

Banca Examinadora:

- Prof. Dr. Jacques Wainer (Orientador)
  Instituto de Computação, Unicamp
- Prof. Dr. Antonio Alfredo Ferreira Loureiro
  Departamento de Ciência da Computação, UFMG
- Prof. Dr. Célio Cardoso Guimarães
  Instituto de Computação, Unicamp
- Prof. Dr. Paulo Lício de Geus (Suplente)
  Instituto de Computação, Unicamp

[1]**Auxílio financeiro da CAPES processo DS-108/00**

Um Algoritmo Guloso para a Configuração e Reconfiguração de *Scatternets* Bluetooth

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Wanderley Ravagnani Junior e aprovada pela Banca Examinadora.

Campinas, 9 de Dezembro de 2003.

Prof. Dr. Jacques Wainer
(Orientador)

Prof. Dr. Edmundo R. M. Madeira
(Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Dedico esta dissertação aos meus pais que sem medir esforços sempre estiveram determinados na minha educação. Dedico também à minha namorada Ana que sempre esteve ao meu lado em todos os momentos. Amo vocês!

Agradecimentos

A Deus por estar sempre comigo e por não deixar faltar força e vontade para que este trabalho se concluísse.

Aos meus orientadores, Prof. Dr. Jacques Wainer e Prof. Dr. Edmundo R. M. Madeira, pela competente orientação, profissionalismo, dedicação e paciência durante estes anos de trabalho.

Aos meus pais, Wanderley e Ana, e aos meus irmãos, Paula e Victor, pelo amor, carinho, incentivo, compreensão e pela força que me deram sempre que necessitei.

Aos meus amigos que me acompanharam durantes esses anos de trabalho. Em especial ao meu amigo Márcio e à minha namorada Ana Teresa.

E finalmente a todos os demais que de alguma forma contribuíram para o desenvolvimento deste trabalho.

Resumo

Bluetooth é uma tecnologia de rádio que permite dispositivos eletrônicos comunicarem a curta distância através de redes *ad-hoc* sem fio denominadas *Scatternets*.

Existem alguns trabalhos que tratam da formação das *Scatternets* Bluetooth, porém nenhum dos trabalhos encontrados levaram em consideração a necessidade de comunicação entre os dispositivos que formam a rede. Isto pode resultar numa perda significativa de desempenho desta rede.

Nesta dissertação de mestrado apresentamos um algoritmo guloso para configurar e reconfigurar *Scatternets* Bluetooth, na tentativa de satisfazer as necessidades específicas de comunicação entre cada par de dispositivos da rede. Este trabalho também mostra algumas simulações para testar a eficiência deste algoritmo.

Abstract

Bluetooth is a radio technology which allows electronic devices to communicate in short distances through ad-hoc wireless networks called Scatternets.

There are some works considering the Bluetooth Scatternet formation, but none of them has considered the communication requirements among the devices that compose the network. This can result in a significant performance loss of the network.

This work presents a greedy algorithm developed to configure and reconfigure Bluetooth Scatternets, in order to satisfy the specific communication requirements of each pair of network devices. This work also shows some simulations to test the efficiency of this algorithm.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Recentemente uma nova tecnologia *wireless* tem sido desenvolvida capacitando dispositivos eletrônicos tais como PCs, notebooks, PDAs, telefones celulares, modems, impressoras, projetores e outros dispositivos, se comunicarem a curta distância. Essa nova tecnologia é denominada Bluetooth, a qual vem ganhando suporte de importantes fabricantes como IBM, Ericsson, Nokia, Toshiba, Microsoft, Intel e muitos outros. Essa tecnologia possibilita diferentes dispositivos de diferentes fabricantes, fixos e/ou móveis, se conectarem formando uma rede *ad-hoc* que permite a transmissão de voz e dados entre esses dispositivos.

A idéia inicial do Bluetooth era basicamente eliminar o uso dos cabos de conexão. Com o andamento do projeto, ficou claro que as aplicações de uma tecnologia desse tipo eram quase ilimitadas. Existem inúmeras propostas para a utilização dessa tecnologia nos mais diversos cenários, entre eles, redes pessoais denominadas PANs (*Personal Area Networks*), a automação industrial, redes de sensores, etc. O Bluetooth possibilita também a criação de redes temporárias de baixo custo, para serem utilizadas, por exemplo, em feiras, conferências e exposições [10].

A tecnologia Bluetooth possibilita a criação de um sistema de rádio, onde os dispositivos se conectam formando uma pequena rede denominada ***piconet***, na qual podem existir oito (8) dispositivos se comunicando a uma taxa máxima de 1 Mbit/s. Além disso, podemos encontrar em ambientes Bluetooth, várias *piconets* sobrepostas, comunicando entre si, formando uma rede *ad-hoc* chamada ***scatternet***. Numa *scatternet*, as *piconets* se comunicam compartilhando um ou mais dispositivos que "saltam"de uma *piconet* para

outra. Esses dispositivos são denominados **pontes**. Já a taxa de comunicação entre as *piconets*, depende do número de pontes e de uma política de escalonamento que define o tempo que uma ponte deve estar ativa em uma *piconet* e quando essa ponte deve "saltar"para a outra *piconet*. A maioria das políticas de escalonamento entre *piconets* testadas por [8, 12] mostrou que esta taxa é aproximadamente de até 100 kbits/s para cada ponte.

A formação de *scatternets* Bluetooth é um assunto muito importante a ser abordado, já que a especificação da tecnologia Bluetooth não trata diretamente deste assunto, ou seja, não diz qual é o protocolo utilizado na criação deste tipo de rede. Além disso, o tempo gasto e o modo como esta rede é formada pode afetar diretamente o desempenho da rede. Alguns trabalhos foram encontrados abordando este assunto [9, 11, 13, 19], onde as *scatternets* são criadas de forma distribuída. Entretanto, estes trabalhos estão preocupados somente com o tempo de formação da *scatternet* e com o número de mensagens trocadas. Porém, estes não levam em consideração a necessidade de comunicação entre os dispositivos que formarão a rede. Isso pode resultar numa perda significativa de desempenho da rede, já que desta forma podemos ter numa mesma *piconet* dispositivos que queiram se comunicar muito pouco e, em *piconets* diferentes, distantes umas das outras, dispositivos que queiram se comunicar a uma alta taxa, como por exemplo, dois laptops que compartilham uma aplicação distribuída. Tendo em vista, que a taxa de comunicação interna de uma *piconet* (1 Mbit/s) pode ser dez (10) vezes maior que a taxa de comunicação entre duas *piconets* (100 kbit/s), seria muito interessante que os dispositivos que necessitam se comunicar a uma alta taxa, ficassem numa mesma *piconet* ou em *piconets* próximas para gerar menos *overhead* de comunicação. Sendo assim, a proposta deste trabalho é configurar uma *scatternet* Bluetooth na tentativa de satisfazer as necessidades específicas de comunicação entre cada par de dispositivos da rede, ou seja, dispor esses dispositivos na rede de maneira que não falte banda de comunicação entre eles. Para tanto, criamos um **Algoritmo Guloso** que tem como objetivo dispor os dispositivos que mais necessitam se comunicar, o mais próximo possível na rede.

Durante este trabalho foram publicados três artigos. No primeiro, [7], partimos de uma restrição muito forte, onde as *scatternet* configuradas seriam totalmente conexas (ou seja, todas as *piconets* estariam conectadas entre si), e a taxa máxima de comunicação entre duas *piconets* era de 100 kbits/s. Além disso, mostramos que a dependência do algoritmo

guloso em relação a taxa de comunicação entre *piconets* era muito pequena, pois quando reduzimos este valor à metade (50 kbits/s), a taxa de satisfação do algoritmo aumentou de 99% para 99,5%. Nos segundo e terceiro artigos, [6, 5] respectivamente, relaxamos as restrições do primeiro, onde as *scatternets* agora devem ser conexas (ou seja, existe pelo menos um caminho ligando duas *piconets* quaisquer da rede), e a taxa de comunicação entre *piconets* é de no máximo 100 kbits/s para cada ponte que conecta duas *piconets*. Estes dois últimos artigos estão refletidos diretamente nesta dissertação de mestrado.

Esta dissertação está estruturada da seguinte maneira. O Capítulo 2 apresenta uma visão geral sobre a tecnologia Bluetooth, o Capítulo 3 refere-se à construção do algoritmo guloso, descrevendo: a modelagem de nossa proposta, o gerador aleatório de *scatternets* Bluetooth (construído para testar a eficiência do nosso algoritmo) e a descrição detalhada do algoritmo guloso proposto. O Capítulo 4 ilustra os resultados obtidos pelo algoritmo na configuração e reconfiguração de *scatternets* Bluetooth, e finalmente o Capítulo 5 apresenta a conclusão e as sugestões para trabalhos futuros.

Capítulo 2

Introdução a Tecnologia Bluetooth

2.1 Visão Geral sobre a Tecnologia Bluetooth

Bluetooth é uma especificação de um padrão aberto para comunicação sem fio, de curto alcance e baixo custo, entre dispositivos através de conexões de rádio. Criada em 1998 por um consórcio das maiores empresas de telecomunicação e computação do mundo (Bluetooth SIG - *Special Interest Group*), a tecnologia Bluetooth permite aos usuários conectar uma ampla variedade de dispositivos fixos (PCs, impressoras, *mouse*, teclados, *scanners*, etc.) e móveis (laptops, PDAs, telefones celulares, etc.) de uma forma bastante simples, sem a necessidade de utilizar cabos de ligação, possibilitando a formação de redes *ad hoc*.

Esse novo padrão visa facilitar as transmissões de dados e voz em tempo real, assegurar a proteção contra interferência e garantir a segurança dos dados transmitidos [15].

O Bluetooth opera na faixa de freqüência ISM (*Industrial, Scientific, Medical*) centrada em 2,4 GHz. Essa faixa de freqüência está disponível mundialmente sem a necessidade de licença e emprega a técnica de saltos de freqüência (*Frequency Hopping*) na transmissão de sinais para combater a interferência [15]. O espectro de freqüência é divido em 79 canais (23 em alguns países), cada um espaçado em 1 MHz.

As camadas física e de enlace da tecnologia Bluetooth estão sendo padronizadas na especificação IEEE 802.15, na tentativa de promover ainda mais esta tecnologia [18].

2.1.1 Redes Bluetooth

A tecnologia Bluetooth possibilita a criação de redes *ad hoc*, onde não há diferença entre as unidades de rádio, ou seja, não há distinção entre estações base e terminais como nas redes convencionais de telefonia celular. Também não há uma infra-estrutura de cabos para suportar a conectividade entre os dispositivos, não há um controlador central para coordenar as interconexões e também não há intervenção de operadores [2]. Além disso, a especificação Bluetooth fornece mecanismos para que os dispositivos possam descobrir uns aos outros sem um conhecimento prévio da localização de cada um, além de permitir a descoberta de serviços que estão disponíveis na rede através do protocolo SDP Bluetooth (*Service Discovery Protocol*) [14].

Piconets

Em ambientes Bluetooth os dispositivos se comunicam formando uma pequena rede denominada ***piconet***, na qual podem existir até oito dispositivos interligados, sendo que esses dispositivos compartilham um mesmo canal de comunicação (*FH/TDD Bluetooth channel*). Para controlar o tráfego e o acesso nesse canal, um desses dispositivos se torna o mestre da *piconet* e os demais se tornam escravos.

O canal Bluetooth é representado por uma seqüência pseudo-aleatória de saltos distribuídos em 79 (ou 23) canais de freqüência. Cada *piconet* possui uma seqüência única de saltos que é determinada pelo endereço Bluetooth do dispositivo mestre e a fase desta seqüência é determinada pelo relógio do mestre [15]. Os escravos devem estar sincronizados ao mestre, sendo assim os escravos utilizam o endereço Bluetooth do mestre para selecionar a mesma seqüência de saltos e adicionam um *Offset* ao seu relógio para sincronizarem-se à mesma freqüência de saltos do mestre [2].

No domínio do tempo, um canal é divido em intervalos com duração de 625 μs chamados ***slots***. Um esquema de *Time Division Duplex* (TDD) é utilizado na transmissão (*fullduplex*) de pacotes, onde mestre e escravo transmitem alternadamente, sendo que cada pacote é transmitido numa freqüência diferente [15]. Esse esquema é ilustrado na Figura 2.1. O mestre sempre inicia suas transmissões em *slots* com numeração par e os escravos em *slots* com numeração ímpar. Os pacotes podem ocupar 1, 3 e 5 *slots* e durante a transmissão de um pacote (mesmo que esse ocupe mais de um *slot*) a faixa de

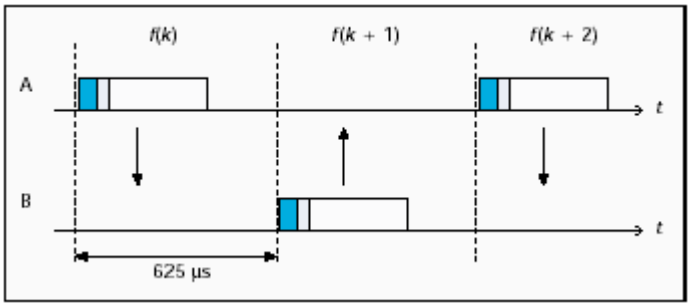

Figura 2.1: Canal FH/TDD no Bluetooth

freqüência permanece fixa e é derivada do valor do relógio do dispositivo mestre, como mostra a Figura 2.2.

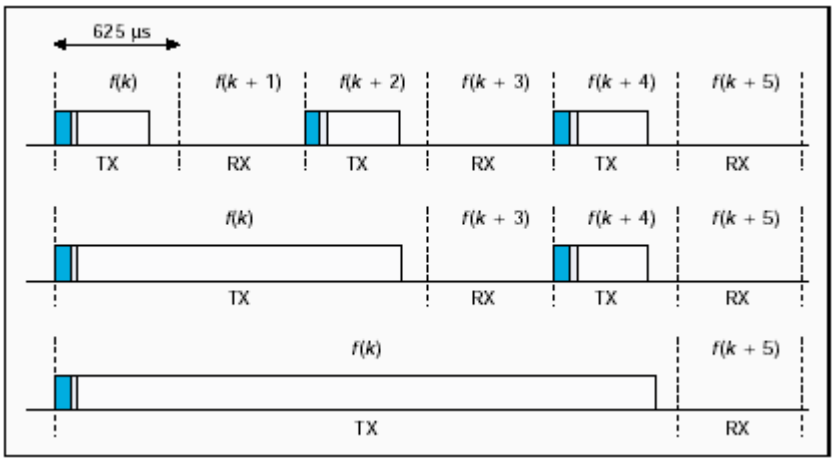

Figura 2.2: Pacotes Multi-*Slots*

Para evitar colisões numa *piconet* devido a múltiplas transmissões de dispositivos escravos, o mestre utiliza a técnica de *polling*, na qual somente um dispositivo pode transmitir por vez. Dessa forma, um escravo somente poderá transmitir em um *slot* caso ele tenha sido endereçado pelo mestre no *slot* anterior.

Scatternets

Tipicamente, em aplicações Bluetooth, várias *piconets* podem se sobrepor ou coexistir numa mesma área, comunicando entre si, formando um sistema *ad hoc* denominado ***scatternet***, onde as *piconets* que compõem este sistema não devem estar sincronizadas [15]. A Figura 2.3 exemplifica os tipos de redes (*piconet* e *scatternet*) formadas entre dispositivos Bluetooth.

Nas *scatternets*, um dispositivo (nó) pode participar de diferentes *piconets* utilizando a técnica de *Time Division Multiplexing* (TDM). Isto é, um nó pode participar seqüencialmente de diferentes *piconets*, sendo que este pode somente estar ativo em uma *piconet* por vez [1]. Sendo assim, um nó pode saltar de uma *piconet* para outra ajustando seus parâmetros de sincronização (relógio e endereço Bluetooth do dispositivo mestre). Um nó também pode mudar de papel quando salta de uma *piconet* para outra, ou seja, um nó pode ser o mestre em uma *piconet* e escravo em outra. O problema neste caso é que quando esse nó deixa uma *piconet* (X) onde ele é o mestre e salta para uma outra (Y), o tráfego da *piconet* X fica suspenso até o seu retorno. Porém, é importante observar que um nó não pode ser o mestre em mais de uma *piconet* devido ao fato da sincronização dentro de uma *piconet* ser feita utilizando os parâmetros do dispositivo mestre. Dessa forma, as *piconets* de uma *scatternet* podem se comunicar compartilhando um (ou mais) nó(s) com as outras *piconets*. O nó compartilhado é denominado **ponte**.

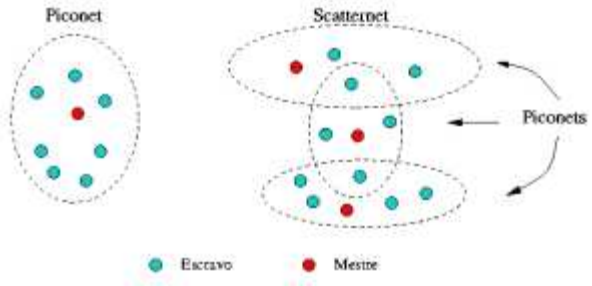

Figura 2.3: Tipos de redes formadas entre dispositivos Bluetooth

Em relação a taxa de comunicação, é possível afirmar que um canal Bluetooth tenha um *throughput* máximo (teórico) de 1 Mbit/s entre os dispositivos pertencentes a uma mesma *piconet* [1, 2, 3]. Porém não é fácil conhecer o valor do *throughput* máximo

suportado por uma ponte (comunicação entre *piconets*), já que é necessário saber o tempo que uma ponte leva para mudar de uma *piconet* para outra e quanto tempo essa ponte fica ativa em cada *piconet*. O valor desse *throughput* depende principalmente de uma política de escalonamento entre *piconets* que além de determinar o tempo que uma ponte fica ativa numa *piconet*, determina também quando uma ponte deve saltar de uma *piconet* para uma outra. Alguns trabalhos abordaram esse assunto propondo diferentes políticas de escalonamento, como é o caso de [4, 8, 12, 16, 17], porém esta ainda é uma questão em aberto pois a especificação Bluetooth ainda não determinou qual política será utilizada. Portanto, não podemos dizer ao certo qual é o valor do *throughput* máximo suportado por uma ponte de uma *Scatternet* Bluetooth.

2.1.2 Pacotes

Como já foi dito, os dados em uma rede Bluetooth são transmitidos em pacotes. O formato geral de um pacote Bluetooth é ilustrado na Figura 2.4.

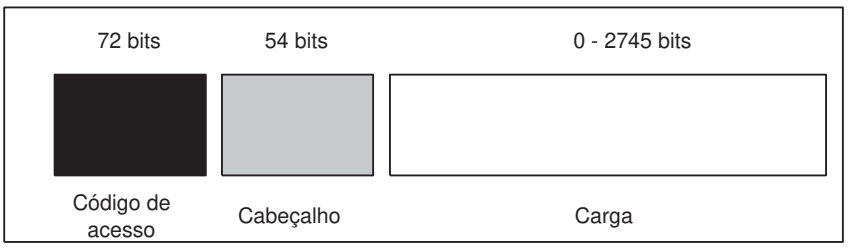

Figura 2.4: Formato Geral de um Pacote Bluetooth

Cada pacote Bluetooth é divido em três partes:

1. Código de Acesso (*Access Code*): possui um tamanho fixo de 72 bits e é utilizado para identificar todos os pacotes transmitidos numa mesma *piconet*. Existem basicamente três tipos de *access code*: *Channel Access Code* (CAC) utilizado para identificar uma *piconet*, *Device Access Code* (DAC) utilizado principalmente para sinalizar o procedimento *Page* que será visto adiante (Seção 2.1.4) e o *Inquiry Access Code* (IAC) que pode ser utilizado por um dispositivo para descobrir quais outros dispositivos estão a seu alcance (procedimento *Inquiry*);

2. Cabeçalho (*Header*): possui um tamanho fixo de 54 bits e é utilizado para controlar as informações de uma conexão, tais como endereço do dispositivo destino na *piconet* (AM_ADDR), tipo do pacote (síncrono ou assíncrono), seqüência, checagem de erros (HEC), etc.

3. Carga (*Payload*): possui um tamanho variável de 0 a 2.745 bits e é utilizado principalmente no transporte de dados.

2.1.3 Tipos de Enlaces

A tecnologia Bluetooth suporta dois tipos diferentes de enlaces:

- ***Synchronous Connection Oriented (SCO)***;
- ***Asynchronous Connectionless (ACL)***.

O SCO é um enlace ponto-a-ponto, simétrico, entre o mestre e um único escravo. Esse enlace utiliza *slots* reservados em intervalos regulares mantidos pelo mestre. Um mestre suporta até três enlaces SCO e um escravo suporta até três enlaces SCO com o mesmo mestre, ou apenas dois com mestres diferentes. Sua principal aplicabilidade é a transmissão de voz. O ACL é um enlace ponto-a-multiponto entre o mestre e todos os escravos de uma *piconet*. Sua principal aplicabilidade é a transmissão de dados [10].

O enlace ACL pode ser usado juntamente com o SCO para suportar aplicações multimídia. Nesse caso, o ACL utiliza *slots* não reservados pelo SCO [3].

2.1.4 Estabelecimento de Conexões

O processo de estabelecimento de conexões entre dispositivos, onde esses dispositivos saem de um estado *Standby* para um estado *Connection*, consiste basicamente de dois procedimentos: ***Inquiry*** e ***Page***. O procedimento *Inquiry* permite um dispositivo (X) descobrir quais dispositivos estão a seu alcance, descobrindo seus endereços Bluetooth e os valores de seus respectivos relógios. Dessa forma esse dispositivo X pode se conectar com alguns dos dispositivos encontrados através do procedimento *page*. Esses procedimentos estão resumidamente descritos a seguir:

O procedimento *Inquiry* é dividido em três estados:

- *Inquiry*: nesse estado um dispositivo que deseja estabelecer uma conexão e não sabe quais são os outros dispositivos que estão em sua área de alcance e suas características, difunde em diferentes freqüências mensagens do tipo *inquiry*. Além disso, esse dispositivo também fica esperando por mensagens do tipo *inquiry response*.

- *Inquire Scan*: nesse estado um dispositivo verifica periodicamente se existe algum outro dispositivo tentando se comunicar, mais precisamente, esperando por mensagens do tipo *inquiry*. Quando esse dispositivo recebe essa mensagem ele passa para o estado *Inquiry Response*.

- *Inquiry Response*: nesse estado o dispositivo que recebeu a mensagem do tipo *inquiry* pode responder com uma mensagem do tipo *inquiry response*, que corresponde a um pacote FHS (pacote especial utilizado para a sincronização entre dispositivos) contendo basicamente seu endereço Bluetooth e o valor de seu relógio.

Esse procedimento está ilustrado na Figura 2.5.

A partir daí, esses dispositivos podem voltar a seus estados originais (*Standby* ou *Connection*) ou entrar no procedimento *Page*. O procedimento *Page* é dividido em quatro estados:

- *Page*: nesse estado um dispositivo (mestre) que deseja estabelecer uma conexão com um outro dispositivo cujo endereço Bluetooth e informações de sincronismo são previamente conhecidos (dispositivo escravo), envia para o dispositivo escravo uma mensagem do tipo *page* contendo o *device access code* (DAC) do dispositivo escravo. Em seguida fica esperando por uma resposta do tipo *page response*. Quando o mestre recebe essa resposta ele passa para o estado *Master Response*.

- *Page Scan*: nesse estado o dispositivo escravo fica esperando por uma mensagem do tipo *page* contendo seu próprio *device access code* (DAC). Quando esse dispositivo recebe essa mensagem ele passa para o estado *Slave Response*.

- *Slave Response*: nesse estado o dispositivo escravo envia uma mensagem do tipo *page response* contendo seu próprio DAC para o mestre. Em seguida o escravo fica esperando um pacote FHS do mestre. Quando ele recebe esse pacote, ele envia uma

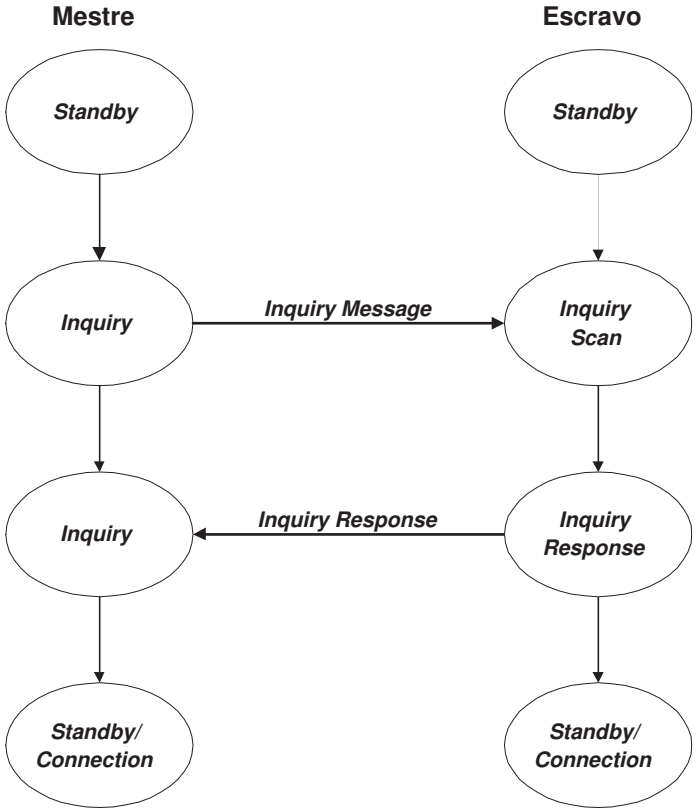

Figura 2.5: Transição de estados durante o procedimento *Inquiry*

resposta para o mestre (ACK) informando que recebeu o pacote FHS. A partir desse momento esse escravo entra no estado *Connection*.

- *Master Response*: nesse estado o dispositivo mestre envia um pacote FHS contendo principalmente seu endereço Bluetooth e o valor de seu relógio (informações para a sincronização entre os dispositivos). Em seguida esse mestre fica esperando por uma reposta do dispositivo escravo (ACK) informando o recebimento do pacote FHS. Ao receber o ACK, o mestre também entra no estado *Connection* e a partir desse momento esses dispositivos estão conectados e utilizam as informações de sincronismo do dispositivo do mestre para se comunicarem.

Esse procedimento está representado na Figura 2.6.

É importante observar que se um dispositivo transmite uma mensagem do tipo *inquiry*

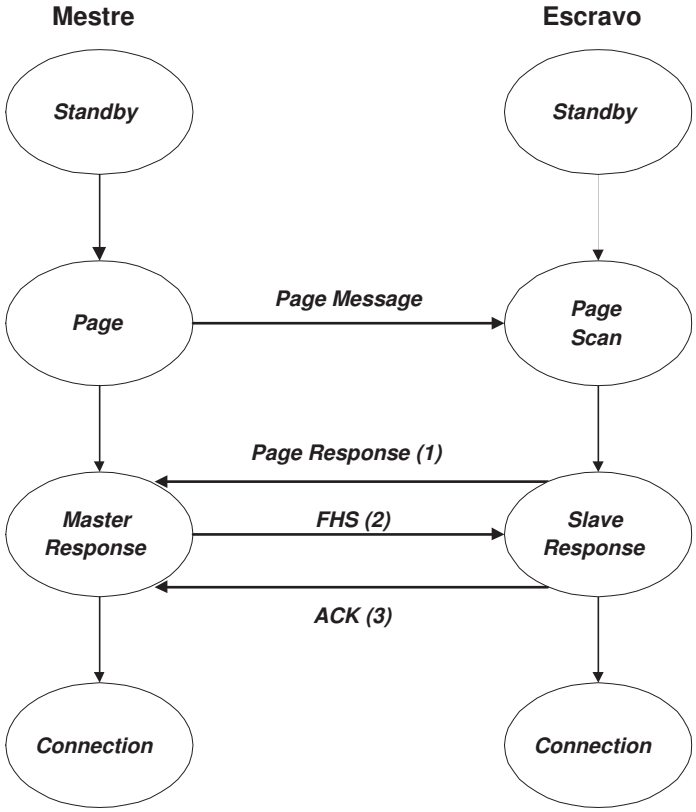

Figura 2.6: Transição de estados durante o procedimento *Page*

ou do tipo *page*, recebendo respostas a essa mensagem, então esse dispositivo entrará no estado *Connection* como mestre. Se um dispositivo fica esperando por mensagens do tipo *inquiry* (*Inquiry Scan*) ou do tipo *page* (*Page Scan*) e responde a ela, então esse dispositivo entrará no estado *Connection* como escravo.

2.2 Trabalhos Relacionados

Os trabalhos relacionados listados aqui tratam do problema da formação de uma *scatternet* Bluetooth. Como já foi dito anteriormente, esses trabalhos não levam em consideração a necessidade de comunicação dos dispositivos que irão formar a rede, sendo que nenhum trabalho foi encontrado abordando esse assunto. Os trabalhos encontrados estão preocupados em mostrar como as *scatternets* são formadas e quão eficiente é essa formação, em

termos de tempo e número de mensagens trocadas entre os dispositivos.

2.2.1 Proposta de Ramachandran [11]

Ramachandran propõe dois algoritmos, um randômico e outro determinístico, para a formação de *scatternets* Bluetooth. Para construir esses algoritmos as seguintes suposições foram feitas:

- Os nós deveriam estar organizados numa topologia estrela dentro de uma *piconet*. Cada nó seria identificado como mestre ou como escravo, onde o mestre é o centro da estrela e os restantes escravos;
- O número máximo de escravos numa *piconet* é sete;
- O tamanho de uma *piconet* deveria ser máximo;
- A troca de papel entre um mestre e um escravo deveria ser evitada, pois essa troca é custosa;
- A transferência de nós de uma *piconet* para outra, depois da conexão estar estabelecida, também é custosa, e deveria ser evitada;
- No final do algoritmo, cada nó sabe se ele é um mestre ou um escravo. Se ele é um escravo, ele sabe quem é o mestre de sua *piconet*;
- A rede deveria estar conectada, e também não haveria nós sem conexão;
- Para terminar, um único nó da *scartternet* deveria ter informações completas sobre todas as *piconets* (super-mestre).

O algoritmo randômico utiliza uma idéia importante, onde um dispositivo fica continuamente no estado *Inquiry* ou continuamente no estado *Inquiry Scan*, na intenção de aumentar a probabilidade de uma mensagem alcançar um outro dispositivo. Tal algoritmo possui dois estágios. O primeiro é randômico e utiliza cálculos probabilísticos para que no final dele, cada nó se torne um *master-designate* ou um *slave-designate*, ou seja, um mestre ou escravo provisório que pode trocar de papel durante o segundo estágio. Porém essa troca é pouco provável de ocorrer, pois esse primeiro estágio tenta alcançar

um número ideal de mestres ($k = \lceil N/8 \rceil$, onde N é o número de nós da rede) para a *scatternet*. O segundo estágio corrige o efeito randômico do primeiro, decidindo o estado final de cada nó, isto é, mestre ou escravo. Um super-mestre é eleito, o qual conta o número atual de mestres e coleta informação sobre todos os nós. O super-mestre pode então executar algum algoritmo centralizado para formação de pontes e também de uma topologia de rede desejada (no caso estrela).

O algoritmo determinístico não utiliza a mesma idéia descrita acima, ou seja, nesse algoritmo os nós alternam entre os estados *Inquiry* e *Inquiry Scan*, o qual aumenta o tempo esperado para que um nó descubra o outro. Entretanto, esse algoritmo é bem mais simples de ser implementado. A idéia básica desse algoritmo é que os nós vão descobrindo uns aos outros e formando uma árvore de repostas, onde a raiz de cada árvore é o mestre da *piconet*. Segundo o autor isso paraleliza a formação das *piconets*. Nesse algoritmo, cada nó i contém uma variável $i.phase$ que armazena o número de resposta às mensagens *Inquiry*. A cada resposta recebida, essa variável é incrementada. Quando a variável $i.phase$ de um nó da árvore for igual a 8, esse nó se elege o mestre e estabelece conexão com todos os outros nós da árvore. Entretanto pode acontecer de mais de um nó ter sua variável $i.phase$ igual a 8. Em tal situação, o mestre que receber mensagens de instrução (as quais são enviadas apenas por nós mestres) de algum de seus escravos, volta ao estado *Inquiry*. A segunda parte desse algoritmo envolve a eleição de um super-mestre que é feita de forma similar a primeira parte desse algoritmo (esquema de árvore de respostas), com a diferença de que os nós envolvidos nessa parte são apenas os nós mestres.

2.2.2 Proposta de Salonidis [13]

Salonidis também propõe um algoritmo para solucionar o problema da formação de *scatternets* e faz as seguintes observações:

- O estabelecimento de conexão na rede deveria ser executado de maneira totalmente distribuída. Isso significa que cada dispositivo começa operando assincronamente e em seu estado inicial ele não tem conhecimento algum sobre os nós que formarão a rede;
- O tempo de formação da rede deveria ser minimizado de forma a ser tolerável pelo usuário final;

- Todo nó deve ser capaz de alcançar todos os outros nós da rede resultante, isto é, a rede deve ser conectada;
- As *piconets* são interconectadas através de pontes na forma escravo/escravo ou mestre/escravo.

Na ausência de alguns critérios de formação da *scatternet*, e na tentativa de projetar um protocolo simples e rápido, os autores propõem as seguintes propriedades que a rede resultante irá satisfazer:

- Uma ponte pode conectar somente duas *piconets* (restrição sobre o grau da ponte), para que esse nó não fique sobrecarregado;
- Dado um número de nós N, a *scatternet* resultante deverá ser constituída de um número mínimo de *piconets*, o que facilita o controle da rede;
- A *scatternet* resultante deverá ser **completamente conectada**, ou seja, todos os mestres deverão estar conectados a todos os outros mestres da rede através de nós pontes;
- Duas *piconets* compartilham apenas uma ponte (restrição sobre a intersecção de *piconets*).

O algoritmo proposto é divido em três fases. A primeira fase consiste na eleição de um coordenador. Tal coordenador eventualmente saberá o número, os identificadores e as informações de sincronismo de todos os nós participantes do processo de construção da rede. Cada nó x tem uma variável chamada *votes* que é inicializada com o valor 1. Depois da inicialização, os nós alternam entre os estados *Inquiry* e *Inquiry Scan*. Quando um nó x descobre um nó y, eles se conectam ponto-a-ponto e comparam suas variáveis *votes*. O vencedor será o que tiver a variável *votes* de maior valor e este incrementará sua variável com o valor da variável *votes* do nó perdedor. Se os valores de suas variáveis forem iguais, o vencedor será o nó que tiver o maior identificador (endereço Bluetooth). O perdedor envia para o vencedor os pacotes FHS (informações de sincronismo) de todos os nós dos quais ele foi o ganhador. Após enviar os pacotes ele entra num estado de *Page Scan*. O coordenador eleito será o vencedor dos primeiros $(N - 1)$ confrontos.

Na segunda fase, o coordenador verifica o número de nós que foi descoberto durante a fase 1. Se o número de nós for menor que 8 ele se conecta a todos os nós que estão no estado de *Page Scan* e uma *piconet* é formada, tendo o coordenador como mestre e o restante dos nós como escravos. Nesse caso especial o algoritmo termina aqui. Se o número de nós for maior que 7, o coordenador calcula o número mínimo P de *piconets* da seguinte forma: $P = \lceil (17 - \sqrt{289 - 8N})/2 \rceil$, onde $1 \leq N \leq 36$. O algoritmo trabalha com essa faixa de N, pois segundo os autores, um grande número de nós pode conduzir a um esquema onde a *scatternet* formada não seja completamente conectada. Depois de calcular P, o coordenador seleciona $(P-1)$ nós além dele como mestres e toma $(P(P-1))/2$ nós como pontes. E conseqüentemente os nós restantes serão os escravos. Dessa forma o coordenador cria, para cada mestre x, duas listas de conectividade, uma de escravos e uma de pontes *(SlaveList(x) e BridgeList(x))* que são constituídas de pacotes FHS e as envia para os respectivos mestres, terminando aí a fase 2.

Durante a fase 3, cada mestre se conecta com seus escravos e pontes definidos em suas duas listas de conectividade. Quando um nó é notificado pelo seu mestre que é uma ponte, ele fica esperando a conexão do segundo mestre. Quando isso acontece, a ponte envia uma notificação de *Connected* para dois mestres. Logo que o mestre recebe essa notificação de suas pontes, uma *scatternet* completamente conectada de P *piconets* é formada e o algoritmo termina.

2.2.3 Outras Propostas

Apresentamos duas outras propostas para solucionar o problema da formação de *scatternets* Bluetooth. A primeira delas foi dada por Ching Law [9]. Em seu trabalho a rede formada deve possuir as seguintes propriedades:

- Qualquer dispositivo pode pertencer no máximo a duas *piconets*. Ou seja, cada ponte deve conectar apenas duas *piconets* para não ficar sobrecarregada;
- O número de *piconets* deve ser próximo do ótimo, ou seja: $\lceil (n-1)/8 \rceil \leq P \leq \lfloor (n-2)/7 \rfloor + 1$.

A única suposição feita por ele é que os dispositivos estão próximos entre si na rede de tal forma que quaisquer dois dispositivos possam se conectar (de 10 a 100 metros).

Esse trabalho propõe um algoritmo simples baseado em procedimentos. O procedimento *Main* é executado pelos dispositivos mestres no começo de cada rodada. Inicialmente todos os dispositivos são considerados mestres. No procedimento *Main* os dispositivos podem chamar o procedimento *Seek* ou o procedimento *Scan* com uma probabilidade de 50%. No procedimento *Seek* os dispositivos tentam encontrar novos escravos (os quais estão executando o procedimento *Scan*) entrando no estado *Inquiry*. Se um novo escravo foi encontrado, ele entra no estado *Page* e finalmente executa o procedimento *Connected*. Já no procedimento *Scan* os dispositivos entram no estado *Inquiry Scan* e após serem contactados por um mestre, entram no estado *Page Scan* e passam a ser escravos. O procedimento *Connected* garante que as propriedades descritas acima sejam respeitadas, agrupando e movendo os dispositivos de uma *piconet* para outra.

O diferencial desse trabalho é que seu algoritmo constrói a rede sem ter conhecimento do número de nós que irão pertencer a ela e também não realiza eleições de líderes e portanto nenhum nó (super-mestre) tem o conhecimento de toda a rede. Esse algoritmo nunca termina, pois os mestres executam o procedimento *Main* de tempos em tempos na expectativa de encontrar novos escravos que possam ser inseridos nessa rede em qualquer momento.

A segunda proposta aplica a tecnologia de agentes móveis para resolver o problema da formação de *scatterntes* Bluetooth [19]. Um ponto interessante a ser destacado neste trabalho é que nem todos os dispositivos precisam estar na mesma área de alcance, pois ele utiliza os agentes para levar as informações de sincronização entre os dispositivos da rede, podendo assim, encontrar uma *piconet* na rede, na qual um novo dispositivo possa ser inserido.

Capítulo 3

Construção do Algoritmo Guloso

3.1 Problema da Formação de *Scatternets* Bluetooth

A formação de uma *scatternet* Bluetooth é um aspecto muito importante a ser ressaltado, pois o tempo e o modo como essa formação é feita podem afetar o desempenho da rede. Esse problema foi tratado por [9, 11, 13, 19], onde estes criam uma topologia de rede de forma distribuída (Ver detalhes na Seção 2.2). Porém, eles não consideram a necessidade de comunicação entre os dispositivos pertencentes à rede, o que pode levar a uma perda de desempenho. Por exemplo, numa rede gerada pelos trabalhos citados acima, onde um laptop e uma impressora, que precisam se comunicar a uma taxa muito alta, podem estar dispostos em *piconets* diferentes, distantes uma da outra, fazendo com que o tráfego dessa *scatternet* fique comprometido, já que os dados transmitidos entre eles terão que percorrer as *piconets* intermediárias até chegarem aos seus destinos. Além disso, como visto anteriormente, a comunicação entre *piconets* pode gerar um retardo no repasse dos dados, pois as pontes necessitam saltar de uma *piconet* para outra, ajustando seus parâmetros de sincronização para repassar os dados. Sendo assim, seria muito interessante que os dispostivos, que mais necessitam se comunicar, permaneçam próximos na rede (ou numa mesma *piconet* ou em *piconets* próximas) na tentativa de gerar menos *overhead* de comunicação nessa rede.

Além disso também devemos considerar cenários onde uma *scatternet* já formada, que satisfaz a necessidade de comunicação entre seus dispositivos, depois de um certo tempo passa a não suportar essas necessidades. Isso pode ocorrer quando um ou mais

dispositivos são inseridos/removidos nessa rede, ou até mesmo quando uma outra *piconet* ou *scatternet* é inserida/removida. Ou ainda pelo simples fato dessa necessidade de comunicação entre os dispositivos pertencentes a essa rede não ser constante. Portanto, para reduzirmos novamente esse *overhead* de comunicação, seria necessário reconstruir essa "nova" *scatternet* na tentiva de voltar a satisfazê-la. Sendo assim, descrevemos esse problema e nossa proposta para resolvê-lo da seguinte forma:

> Devido às restrições de largura de banda e tendo em vista a necessidade de um dispositivo i qualquer, pertencente a uma *scatternet* S, comunicar com qualquer outro dispositivo j pertencente a mesma *scatternet* S a uma taxa T_{ij}, a proposta desse trabalho é expor e mostrar a eficiência de um algoritmo que projetamos para configurar essa rede Bluetooth (*scatternet* S) na tentativa de satisfazer (no caso da configuração), ou voltar a satisfazer (no caso da reconfiguração), a necessidade de comunicação (T_{ij}) de todo dispositivo i, j, onde $i \neq j$. A idéia desse algoritmo é dispor os dispositivos na rede de tal forma que os que mais necessitam comunicar entre si permaneçam o mais próximo possível na rede (Algoritmo Guloso).

3.2 Modelagem

Devido à complexidade do problema de formação de *scatternets* Bluetooth considerando a necessidade de comunicação entre seus dispositivos, descrito na Seção 3.1, algumas restrições foram feitas para modelar nossa proposta de configuração e reconfiguração de *scatternets* Bluetooth de forma a satisfazer as necessidades específicas de comunicação existentes entre cada par de dispositivos pertencentes à rede:

- A *scatternet* possui N dispositivos, e os N dispositivos estão próximos entre si, de forma que quaisquer dois dispositivos possam se comunicar;
- A *scatternet* é **conexa**, ou seja, existe pelo menos um caminho ligando quaisquer duas *piconets* da *scatternet* através de pontes;
- Uma ponte deve conectar somente duas *piconets*, pois assim ela não fica sobrecarregada. Se uma ponte pertencer a várias *piconets*, então ela poderá se tornar um

gargalo para a comunicação entre essas *piconets*, já que essa ponte pode ficar ativa somente em uma *piconet* por vez [9];

- Como o *throughput* entre *piconets* ainda é uma questão aberta, nós assumimos que o *throughput* suportado por uma ponte que conecta quaisquer duas *piconets* é no máximo de 100 kbits/s, pois os trabalhos [8, 12] mostraram que para a maioria das políticas de escalonamento por eles testadas, esse valor é de aproximadamente 100 kbits/s.

Com os itens acima descritos, poderemos agora mostrar como geramos as redes para testar a eficiência do algoritmo guloso, bem como expor em detalhes como esse algoritmo funciona.

3.3 Gerador de Redes Aleatórias Bluetooth

Em nosso projeto desenvolvemos um gerador de redes aleatórias Bluetooth, onde essas redes são de alguma forma factíveis (ou seja, cada rede possui pelo menos uma solução) em relação às necessidades de comunicação entre os dispositivos da rede. As redes geradas servirão para testar a eficiência do algoritmo guloso.

Três topologias diferentes de rede foram utilizadas:

- **Triângulo-Aresta**: uma rede com a seguinte estrutura definida: um triângulo, seguido de uma aresta, seguido de um triângulo, seguido de uma aresta, e assim consecutivamente. Nesta topologia cada *piconet* possui um número definido, porém diferente, de pontes. Esta topologia de rede foi utilizada, pois sua estrutura assemelha-se com a idéia gulosa do algoritmo, como será visto no Capítulo 4;
- **Anel**: uma rede circular onde cada *piconet* possui exatamente duas pontes;
- **Aleatória**: uma rede que pode ter diferentes topologias. Entretanto assumimos que nesta topologia aleatória, cada *piconet* possuirá no mínimo uma e no máximo três pontes, pois utilizando mais que três pontes por *piconet*, degradaríamos o seu desempenho, já que cada ponte fica ativa em uma só *piconet* por vez [5, 6].

Estas três topologias estão respectivamente exemplificadas na Figura 3.1 em forma de grafo, onde os vértices representam as *piconets* e as arestas as pontes.

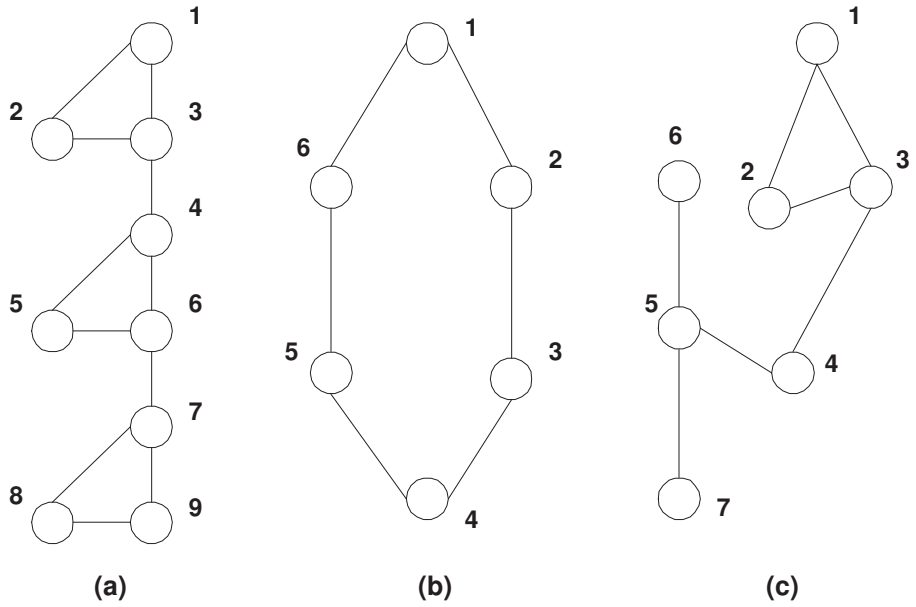

Figura 3.1: Topologias utilizadas: (a) Triângulo-Aresta, (b) Anel, (c) Aleatória

O algoritmo para gerar um grafo (rede) aleatório com no máximo três arestas (pontes) por vértice (*piconet*) funciona da seguinte forma:

1. Para cada vértice v, define-se o número de arestas que este possuirá (na). O número de arestas na de v é um número aleatório entre 1 e 3. O próximo passo é conectar os vértices para formar o grafo;

2. Para cada vértice v, escolhe-se outros na vértices para conectá-los à v. Cada um dos vértices escolhidos deve possuir no máximo 2 arestas, pois se possuir 3 arestas estaremos extrapolando o número máximo de arestas permitido, já que para cada vértice escolhido, aumentamos o seu número de arestas em uma unidade. Fica claro, que as vezes não é possível encontrar outros na vértices, pois os demais vértices do grafo podem já estar carregados ($na = 3$), ou porque não temos outros na vértices no grafo. Quando isto acontece, o número de arestas do vértice v é igual ao número de vértices que foi possível encontrar no grafo;

3. Em seguida, executamos um algoritmo de busca em largura (*Breadth-First Search* - BFS) para verificar se o grafo gerado é conexo. Este algoritmo parti de um vértice qualquer s e descobre todos os vértice que são alcançáveis a partir de s. Se um vértice não for alcançável a partir de s, significa dizer que o grafo não é conexo. A recíproca é verdadeira. O algoritmo de busca em largura funciona da seguinte forma:

   (a) Inicialmente colorimos de branco todos os vértices do grafo;

   (b) A seguir, escolhemos um vértice s qualquer como sendo a origem da nossa busca e o inserimos na fila Q (*first-in, first-out*), inicialmente vazia;

   (c) Enquanto a fila Q não estiver vazia, escolhemos u como sendo o primeiro elemento da fila Q;

      i. Para cada vértice (d) adjacente à u, analisamos sua cor. Se for branca, inserimos d em Q;

      ii. Quando todos os vértices adjacentes à u forem analisados, removemos u da fila Q e o colorimos de preto;

   (d) Quando a fila Q estiver vazia o algoritmo de busca em largura termina.

4. Em seguida verificamos se todos os vértice foram coloridos de preto. Se foram, então podemos garantir que o grafo gerado é conexo, ou seja, existe pelo menos um caminho ligando quaisquer dois vértices deste grafo. Se pelo menos um vértice tiver a cor branca, então o grafo gerado não é conexo e voltamos ao passo 1 deste algoritmo até encontrarmos um grafo conexo.

O gerador de redes aleatórias Bluetooth funciona da seguinte maneira:

1. Primeiramente escolhemos uma das três topologias, e o número de *piconets* (NP) que a rede Bluetooth possuirá. Se a topologia de rede escolhida foi a aleatória, executamos o algoritmo descrito anteriormente para construirmos um grafo aleatório e conexo com NP vértices;

2. Em seguida, para cada *piconet* definimos o seu número de pontes. Este número depende da topologia escolhida;

3. Depois de definirmos o número de pontes de cada *piconet*, preenchemos as NP *piconets* aleatoriamente com no mínimo um (excluindo as pontes) e máximo oito (incluindo as pontes) dispositivos por *piconet*.

O próximo passo é criar uma estrutura para armazenarmos as necessidades de comunicação dos dispositivos na rede. Para tanto, utilizamos uma matriz triangular inferior de *throughputs* (MT), onde para cada par de dispositivos i, j (onde $i > j$ e $i \neq j$), $MT[i][j]$ significa o quanto i e j necessitam se comunicar na rede (tráfego agregado), o que chamamos de ***throughput* desejado** entre i e j. Finalmente preenchemos a matriz MT da seguinte forma:

- Para cada *piconet* p, calculamos o *throughput* utilizado dentro da *piconet* (***throughput intra-piconet***) e o *throughput* utilizado entre a *piconet* p e as demais *piconets* da rede (***throughput extra-piconet***):

  ***Throughput extra-piconet* de p:** Será um valor aleatório entre 0 e 100 kbits/s vezes o número de pontes da *piconet* p. Esse valor é distribuído aleatoriamente entre os enlaces que ligam os dispositivos pertencentes a *piconet* p a todos os outros dispositivos que não pertencem a p. Os enlaces que ligam os dispositivos de p às pontes da mesma *piconet* p também são considerados como *extra-piconet*;

  ***Throughput intra-piconet* de p:** Será um valor aletório entre 100 kbits/s e 1 Mbit/s menos o *throughput extra-piconet* de p. Este valor é distribuído aleatoriamente entre os enlaces que ligam os dispositivos pertencentes a mesma *piconet* p, exceto aos dispositivos que são pontes em p.

Com isso, criamos aleatoriamente uma *scatternet* Bluetooth que possui pelo menos uma solução. A matriz de *throughputs* desejados (MT) e o número de dispositivos gerados na rede (N) constituem a entrada do algoritmo guloso que será apresentado a seguir.

É importante observar que a simetria de tráfego entre i e j (onde $i \neq j$), em situações reais, não existe, mas isso não faz com que os algoritmos descritos aqui percam a generalidade, já que a matriz de *throughputs* desejados (MT) poderia ser totalmente preenchida e utilizada pelos algoritmos [5, 6, 7].

Exemplo Ilustrativo

Para exemplificar a execução do gerador de redes aleatórias Bluetooth, construiremos uma *scatternet* com apenas três (3) *piconets* e com a topologia anel.

Primeiramente definimos o número de dispositivos de cada *piconet* de forma aleatória (com no máximo oito (8) e no mínimo dois (2) dispositivos, já que numa rede com a topologia anel cada *piconet* possuirá exatamente duas (2) pontes). Então as *piconets* ficaram assim:

A *Piconet* 1 possuirá quatro (4) dispositivos, a *Piconet* 2 possuirá oito (8) e a *Piconet* 3 possuirá seis (6) dispositivos. Assim, o número total de dispositivos dessa rede é quinze (15), pois cada *piconet* possuirá um (1) dispositivo em comum (ponte). Em seguida, distribuímos esses quinze (15) dispositivos aleatoriamente nas *piconets*. Os dispositivos escolhidos como pontes foram o 8, 6 e 13. Veja a Figura 3.2.

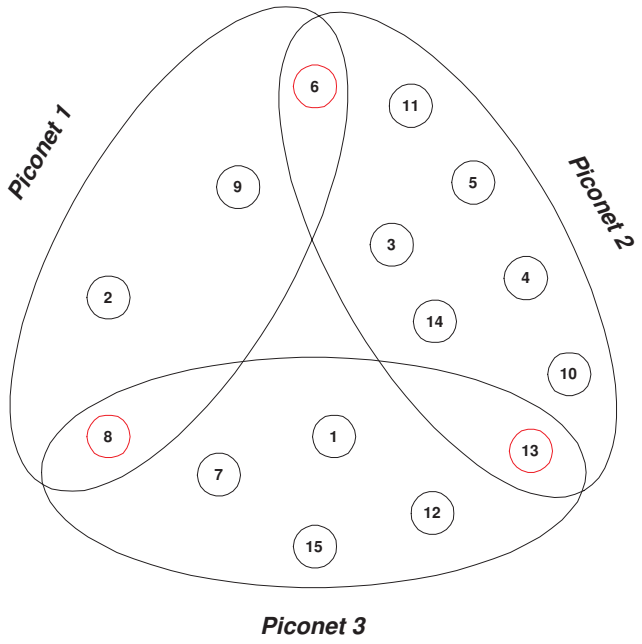

Figura 3.2: Exemplo de uma *scatternet* com três (3) *piconets* gerada aleatoriamente

Depois, calculamos aleatoriamente o valor do *throughput intra-piconet* e *extra-piconet*

para cada *piconet* gerada. Como cada *piconet* possui duas (2) pontes, então o valor do *throughput extra-piconet* de cada uma será de no máximo 200 kbits/s. Já o *throughput intra-piconet* será um valor aleatório entre 100 kbits/s e 1 Mbit/s menos o valor de seu *throughput extra-piconet*. A Tabela 3.1 mostra os valores encontrados.

	Throughput intra-piconet	***Throughput extra-piconet***
***Piconet* 1**	505 kbits/s	112 kbits/s
***Piconet* 2**	433 kbits/s	115 kbits/s
***Piconet* 3**	941 kbits/s	16 kbits/s

Tabela 3.1: *Throughputs* gerados aleatoriamente

Para terminar, para cada *piconet*, distribuímos aleatoriamente os *throughputs intra-piconet e extra-piconet* gerados. A matriz MT, que representa essa distribuição, está representada na Tabela 3.3.

Dispositivos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
2	0													
3	0	7												
4	0	12	21											
5	0	2	75	54										
6	0	0	4	0	0									
7	58	0	1	0	1	0								
8	1	0	0	0	0	0	0							
9	0	505	13	5	0	0	1	0						
10	0	10	22	2	20	2	0	0	1					
11	0	14	45	26	34	0	0	0	19	22				
12	113	0	0	1	0	0	151	0	1	0	1			
13	0	3	0	0	0	3	0	0	0	0	0	0		
14	0	0	48	23	19	0	0	1	15	15	7	0	0	
15	404	0	0	0	0	0	195	0	1	0	0	20	0	1

Tabela 3.2: Matriz de *throughputs* desejados

3.4 Algoritmo Guloso

A idéia do nosso algoritmo guloso é colocar os dispositivos que mais precisam se comunicar, dentro de uma mesma *piconet*, pois o canal *intra-piconet* possui o maior *throughput*. Se isso não for possível, esses dispositivos são colocados em *piconets* próximas, sendo que quanto mais próximas eles estiverem na rede, menos tráfego entre *piconets* será gerado, já que as pontes tendem a ser os gargalos da rede, pois essas ficam alternando de uma *piconet* para outra em intervalos de tempo. O algoritmo recebe como entrada o número de dispositivos (N) e a matriz de *throughputs* desejados (MT). A topologia de rede escolhida para ilustrar o algoritmo é a Triângulo-Aresta. O pseudocódigo do algoritmo é apresentado na Figura 3.3 e seus passos são citados a seguir:

1. Para cada dispositivo i, cria-se uma fila F_i contendo os outros $(N-1)$ dispositivos, em ordem decrescente de acordo com os *throughputs* desejados entre i e os demais dispositivos da rede, para sabermos com quem i deseja se comunicar mais. Também cria-se um ponteiro P_i, inicialmente nulo, que apontará para o primeiro elemento da fila do dispositivo i. Quando o dispositivo i for o escolhido para entrar na rede, o dispositivo apontado por P_i será o seu candidato a ser o próximo elemento a entrar na rede. Este passo é ilustrado pelas linhas 1 a 16 da Figura 3.3. Desta forma, colocamos os dispositivos que mais desejam se comunicar próximos uns dos outros;

2. As pontes escolhidas são os dispositivos que necessitam comunicar muito pouco com os demais dispositivos da rede, pois essas alternam de uma *piconet* para outra em intervalos de tempo. Elas são armazenadas numa fila de pontes (*bridges*) e serão inseridas na rede no final do algoritmo. Aqui nós definimos as pontes como sendo os dispositivos que necessitam comunicar menos que 10 kbits/s com os demais dispositivos da rede. Esse valor foi escolhido pois se escolhermos mais pontes que o necessário para formar a rede, podemos criar novas *piconets* inserindo nessas as pontes restantes, não afetando assim o tráfego da rede, já que cada *piconet* pode ter no máximo oito dispositivos com no mínimo uma ponte, e utilizando esse valor, não ultrapassamos o limite de 100 kbits/s que é o *throughput extra-piconet* máximo para *piconets* com apenas uma ponte (linhas 17 a 26 da Figura 3.3);

3. O primeiro dispositivo i a ser escolhido para entrar na rede é aquele que deseja se comunicar mais com algum outro dispositivo da rede (linha 28). Então criamos a

primeira *piconet* e o inserimos lá. Em seguida removemos i de todas as outras filas e fazemos seu ponteiro P_i apontar para o primeiro elemento de sua fila F_i, pois P_i apontará para o seu candidato a ser o próximo a entrar na rede. Também definimos o número de pontes (lim_j) que essa *piconet* possuirá (de acordo com a topologia, no caso, triângulo-aresta), bem como o número máximo de dispositivos não-pontes ($8 - lim\_j$) dessa *piconet* (linhas 27 a 39);

4. A partir daí, a escolha dos outros dispositivos a entrar na rede é feita da seguinte forma (linhas 40 a 98 da Figura 3.3):

   (a) Para cada dispositivo i, analisamos o seu candidato (c_i) que é apontado por P_i. O candidato c_i a ser escolhido deve respeitar o seguinte critério: incluindo c_i na *piconet* em questão (np), a soma do *throughput* interno de np deve ser menor ou igual a 1 MBit/s (linhas 43 a 72). Se esse critério foi respeitado, verificamos se a condição abaixo é satisfeita:

      i. Inserindo c_i em np, o *throughput extra-piconet* de np passa a ser menor que seu *throughput extra-piconet* máximo (100 kbits/s vezes o número de pontes em np) e o *throughput intra-piconet* também passa a ser menor que seu limite máximo (1 MBit/s menos seu *throughput extra-piconet*). Porém essa condição só será aceita, se a posição de c_i em np (sendo que aqui uma *piconet* é representada matematicamente como um vetor) for maior ou igual a um valor de quebra Lim_{np} correspondente a *piconet* np (**condição de quebra**). Se for menor, guardamos essa posição em Lim_prox_{np} para realizarmos posteriormente o método de *BackTracking* (linhas 50 a 59). Esse valor de quebra Lim_{np}, bem como a necessidade de seu uso, será explicada mais adiante.

   (b) O dispositivo escolhido, d_e, é aquele que obedeceu o critério acima citado *(item a)* e a condição de quebra *(item i)*. Se somente a condição de quebra não foi satisfeita, então d_e será o dispositivo que mais deseja se comunicar com os demais dispositivos já inseridos em np, além de respeitar o *item a* (linhas 60 a 69). Ao inserirmos d_e na *piconet* np, iniciamos sua fila F_{de} fazendo P_{de} apontar para o primeiro elemento de F_{de}, além disso atualizamos todas as outras filas excluindo das tais, o dispositivo d_e escolhido (linhas 73 a 86);

(c) Se nenhum candidato foi escolhido, ou se o número de dispositivos inseridos na *piconet np* for igual a $(8 - lim\_j)$, ou se ocorreu uma condição de quebra, encerramos essa *piconet np* e criamos uma nova (linhas 87 a 90). Além disso, se ainda restam dispositivos a serem inseridos na rede, calculamos o número de pontes e o número máximo de dispositivos não-pontes $(8 - lim\_j)$ para a próxima *piconet* (linhas 91 a 96) e retornamos para o *item a*;

5. Quando o número de dispositivos inseridos for igual a N menos o número de pontes escolhidas, ou quando esgotam-se as pontes escolhidas para formar a topologia da rede (linha 98), inserimos o restante dos dispositivos, se existirem, no final da fila das pontes (*bridges*), e em seguida inserimos as pontes na rede e o restante dos dispositivos são inseridos em novas *piconets* adicionadas no final da rede, encerrando-se assim o algoritmo (linhas 99 a 113 da Figura 3.3).

---

procedimento Guloso(int $lim[]$**, int** $lim\_prox[]$**);**

1: $Bridges \leftarrow \varnothing$; // Fila de pontes
2: **para** cada $i \in S$ **faça**
3: $F[i] \leftarrow \varnothing$; // Fila de candidatos de i
4: $P[i] \leftarrow \varnothing$; // Aponta para o 1º candidato da fila de i
5: $disp[i] \leftarrow N\_UTIL$; // Dispositivo não utilizado
6: $extra\_disp[i] \leftarrow 0$; // Guarda o *throughput extra-piconet* de i
7: **fim para;**
8: // Insere os dispositivos nas filas e calcula os *throughputs extra-piconets*
9: **para** cada $i \in S$ **faça**
10: **para** cada $j \in S$ **faça**
11: **se** $i \neq j$ **então**
12: $extra\_disp[i] \leftarrow extra\_disp[i] + Throughput(i, j)$;
13: Insere_Fila(F[i], Throughput(i, j), j);
14: **fim se;**
15: **fim para;**
16: **fim para;**
17: // Escolhe as pontes de S
18: $n\_bridges \leftarrow 0$;

19: **para** cada $i \in S$ **faça**
20: **se** $extra\_disp[i] \leq 10Kbits/s$ **então**
21: Insere_Fila(Bridges, extra_disp[i], i);
22: $disp[i] \leftarrow BRIDGE$; // Dispositivo ponte
23: $F[i] \leftarrow \varnothing$; $n\_bridges++$;
24: Remove_El(F, i); // Remove i de todas as outras Filas
25: **fim se;**
26: **fim para;**
27: // Dispositivo que possui o maior *throughput* desejado em S
28: $maior\_i \leftarrow MAX(S)$;
29: // Insere maior_i em S
30: $np \leftarrow 0$; $j \leftarrow 0$; $n\_aux \leftarrow 1$;
31: $S.pico[np].elemento[j++] \leftarrow maior\_i$;
32: $disp[maior\_i] \leftarrow UTIL$; // Dispositivo inserido em S
33: $P[maior\_i] \leftarrow F[maior\_i].Prim$;
34: // Para Topologia Triângulo-Aresta
35: $S.pico[np].n\_bridges \leftarrow 2$; // Nº de pontes da *piconet* np
36: $bridges\_usadas \leftarrow 2$; $lim\_j \leftarrow 2$;
37: $throughput\_extra \leftarrow (100Kbits/s * 2)$; // Valor máximo
38: // Remove maior_i de todas as outras filas
39: Remove_El(F, maior_i);
40: // Seleciona os dispositivos a entrar na rede
41: **repita**
42: $escolheu \leftarrow$ **falso**; $quebra \leftarrow$ **falso**; $maior \leftarrow -1$;
43: **para** cada $i \in S$ **e** $j < (8 - lim\_j)$ **faça**
44: **se** $P[i] \neq \varnothing$ **então**
45: $S.pico[np].elemento[j] \leftarrow P[i].id$;
46: // Calcula o *extra-piconet* da *piconet* np
47: $extra \leftarrow Calcula\_Extra(S.pico[np], j)$;
48: $intra \leftarrow Soma\_Intra(S.pico[np], j)$;
49: **se** $intra \leq 1Mbit/s$ **então**
50: **se** $extra \leq throughput\_extra$ **e** $intra \leq (1MBit/s - extra)$ **então**
51: **se** $j \geq lim[np]$ **então**

52: $escolheu \leftarrow$ **verdadeiro**;
53: $quebra \leftarrow$ **verdadeiro**;
54: $maior\_i \leftarrow P[i].id$;
55: **senão se** $j < 8 - lim\_j - 1$ **então**
56: // Utilizado p/ o *BackTracking*
57: $lim\_prox[np] \leftarrow j$;
58: **fim se;**
59: **fim se;**
60: // Calcula quanto o Candidato[i] quer falar
61: // com os demais dispositivos da piconet np
62: $soma\_rel \leftarrow Soma\_Rel(S.pico[np], j)$;
63: // Guarda o melhor candidato,
64: // quando não é possível realizar a quebra
65: **se** $quebra =$ **falso e** $maior < soma\_rel$ **então**
66: $maior \leftarrow soma\_rel$;
67: $escolheu \leftarrow$ **verdadeiro**;
68: $maior\_i \leftarrow P[i].id$;
69: **fim se;**
70: **fim se;**
71: **fim se;**
72: **fim para;**
73: // Se alguém foi escolhido: Insere-o em S.
74: **se** escolheu **então**
75: $S.pico[np].elemento[j] \leftarrow maior\_i$;
76: $disp[maior\_i] \leftarrow UTIL$; // Dispositivo inserido em S
77: $P[maior\_i] \leftarrow F[maior\_i].Prim$;
78: n_aux++; $j++$;
79: Remove_El(F, maior_i);
80: // Ajusta os ponteiros dos Candidatos
81: **para** cada $i \in S$ **faça**
82: **se** $P[i] \neq \varnothing$ **então**
83: $Candiato[i] \leftarrow F[i].Prim$;
84: **fim se;**

```
85:       fim para;
86:     fim se;
87:     se escolheu = falso ou n_aux = N − n_bridges ou quebra = verdadeiro então
88:       // Cria uma nova piconet
89:       S.pico[np].n_elementos ← j;
90:       np + +; j ← 0;
91:       // Se ainda restam dispositivos a serem inseridos em S
92:       se n_aux < N − n_bridges então
93:         // Para topologia Triângulo-Aresta, calcula-se o nº de
94:         // dispositivos pontes e não-pontes p/ a próxima piconet
95:         Calc_n_Pontes(N, n_aux, np, S, lim_j, bridges_usadas, throughput_extra);
96:       fim se;
97:     fim se;
98: até (n_aux < N − n_bridges e S.pico[np].n_bridges > 0)
99: // Descarrega o restante dos dispositivos, se existirem, no final da fila das pontes
100: se n_aux < N − n_bridges então
101:   para cada i ∈ S faça
102:     // Se i não foi utilizado
103:     se bridge[i] = N_UTIL então
104:       Insere_Fila(Bridges, extra_disp[i], i);
105:       n_bridges + +;
106:     fim se;
107:   fim para;
108: fim se;
109: // Insere as pontes em S de acordo com a topologia.
110: // Os dispositivos que restarem são inseridos em novas piconets,
111: // adicionadas no final da scatternet S
112: Insere_Pontes(S, Bridges);
113: Fim.
```

Figura 3.3: Pseudocódigo do Algoritmo Guloso

A complexidade deste algoritmo guloso é $O(N^3)$, pois para cada dispositivo escolhido para integrar a rede (onde são escolhidos N dispositivos), percorre-se todas as N filas de tamanho $(N-1)$, removendo esse dispositivo escolhido. É importante observar que esse é um algoritmo de satisfação que utiliza uma heurística gulosa para tentar satisfazer a necessidade de comunicação da rede.

A condição de quebra (Lim_{np}), acima citada, é muito importante, pois ela limita o número de dispositivos dentro de cada *piconet* p_i, já que sem ela podemos inserir nessa *piconet* alguns dispositivos que querem se comunicar muito pouco com os dispositivos já escolhidos. Isso pode fazer com que uma nova *piconet* p_j seja criada e os próximos dispositivo inseridos nela talvez queiram se comunicar muito com o último dispositivo inserido em p_i, podendo assim extrapolar os *throughputs extra-piconet* de p_i e de p_j. Essa quebra possibilita colocar o último dispositivo de p_i e os dispositivos de p_j numa mesma *piconet*.

Para encontrarmos um valor de quebra adequado para cada *piconet*, o procedimento descrito na Figura 3.4 é executado. Este procedimento segue os seguintes passos: inicialmente geramos a rede através do algoritmo guloso com o valor de quebra (Lim_{np}) igual a oito (8) para todas as *piconets* (linhas 1 a 6 da Figura 3.4). Isso faz com que as *piconets* da rede fiquem carregadas (cheias). O algoritmo guloso retorna a última posição onde poderia ocorrer uma quebra anterior a Lim_{np} (Lim_prox_{np}) para cada *piconet* da rede. Em seguida, vamos reduzindo o valor de quebra de Lim_{np} para Lim_prox_{np} e gerando a rede gulosa, para cada *piconet* que possui seu *throughput intra-piconet* ou *extra-piconet* acima do seu limite máximo (linhas 10 a 16). Se depois de analisarmos todas as *piconets*, a necessidade de comunicação da rede ainda não foi satisfeita, nós retornamos a última *piconet* analisada que ainda pode ser quebrada ($\text{Lim}_{np} \neq \text{Lim\_prox}_{np}$) e novamente vamos reduzindo o valor de quebra e gerando a rede gulosa para cada *piconet* abaixo dela (linhas 17 a 29 - Método de *BackTracking*). Esse procedimento termina quando a necessidade de comunicação da rede for totalmente satisfeita (linha 9) ou quando não encontramos mais nenhuma *piconet* a ser quebrada (linha 23 da Figura 3.4).

---

procedimento Gera_Rede;

1: **para** $i \leftarrow 0$ **até** NP **faça**
2: $\quad lim[i] \leftarrow 8$;

```
3:    lim_prox[i] ← 8;
4:  fim para;
5:  Guloso(lim, lim_prox); // Gera a rede Gulosa
6:  Copia_Rede(S, melhor_S);
7:  i ← 0;
8:  // Enquanto a rede não for satisfeita
9:  enquanto S.pior < 0 faça
10:   enquanto (Throughput_Extra(S, i) > (100kbits/s * S[i].n_bridges)
      ou Throughput_Intra(S, i) > (1Mbit/s − Throughput_Extra(S, i)))
      e lim[i] ≠ lim_prox[i] faça
11:     lim[i] ← lim_prox[i];
12:     Guloso(lim, lim_prox); // Gera a rede Gulosa
13:     se S.pior > melhor_S.pior então
14:       Copia_Rede(S, melhor_S); // Guarda a melhor rede
15:     fim se;
16:   fim enquanto;
17:   se S.pior < 0 e i ≥ NP − 1 então
18:     enquanto i > −1 e lim[i] = lim_prox[i] faça
19:       lim[i] ← 8; lim_prox[i] ← 8;
20:       i − −;
21:     fim enquanto;
22:     se i = −1 então
23:       break; // Termina o procedimento
24:     fim se;
25:     lim[i] ← lim_prox[i];
26:     Guloso(lim, lim_prox); // Gera a rede Gulosa
27:     se S.pior > melhor_S.pior então
28:       Copia_Rede(S, melhor_S); // Guarda a melhor rede
29:     fim se;
30:   senão
31:     i + +;
32:   fim se;
33: fim enquanto;
```

34: **Fim.**

Figura 3.4: Pseudocódigo do procedimento que gera a rede

Utilizando este procedimento a complexidade total do algoritmo passa a ser exponencial, pois ele utiliza o método de *BackTracking* citado acima. Entretanto, como será visto na próximo capítulo, limitaremos o seu tempo de execução em 1,5 segundos, pois neste tempo o procedimento resolve grande parte dos casos.

Se a topologia de rede escolhida for anel, o número de pontes por *piconet* passa a ser fixo (igual a 2) e conseqüentemente o *throughput extra-piconet* máximo de cada *piconet* é de 200 kbits/s.

Exemplo Ilustrativo

Para exemplificar a execução do algoritmo guloso, utilizaremos os dados do exemplo ilustrativo (com 15 dispositivos) do gerador de redes aleatórias Bluetooth (Seção 3.3) como entrada do algoritmo. A topologia de rede utilizada para esse exemplo é indiferente, já que tanto a triângulo-aresta quanto a anel possuem a mesma estrutura para *scatternets* com até três 3 *piconets*.

Inicialmente construímos a *scatternet* com as *piconets* totalmente carregadas (posição de quebra para todas as *piconets* igual a 8). E assim iniciamos a execução do algoritmo guloso.

Em primeiro lugar, construímos as filas dos candidatos a entrar na rede para cada dispositivo e também a fila de pontes. A Tabela 3.3 mostra os dispositivos escolhidos como pontes (dispositivos que desejam comunicar menos que 10 kbits/s com os demais dispositivos da rede) e a Tabela 3.4 ilustra as filas dos candidatos a entrar na rede de cada dispositivo (excluindo as pontes).

Dispositivos (Id)	6	13	8

Tabela 3.3: Fila de pontes

Em seguida, criamos a primeira *piconet* (*piconet* 1) e inserimos nela o dispositivo 2, já

que este é o dispositivo que mais deseja se comunicar com um outro dispositivo, no caso o 9. Logo após, removemos o dispositivo 2 de todas as filas de candidatos, pois este já foi inserido na rede. Assim, o primeiro dispositivo da fila de candidatos do dispositivo 2 é candidato a entrar na rede.

Candidatos	Id	15	12	7	14	11	10	9	2	5	4	3
de 1	kbits/s	404	113	58	0	0	0	0	0	0	0	0
Candidatos	Id	9	11	4	10	3	5	15	14	12	7	1
de 2	kbits/s	505	14	12	10	7	2	0	0	0	0	0
Candidatos	Id	5	14	11	10	4	9	2	7	15	12	1
de 3	kbits/s	75	48	45	22	21	13	7	1	0	0	0
Candidatos	Id	5	11	14	3	2	9	10	12	15	7	1
de 4	kbits/s	54	26	23	21	12	5	2	1	0	0	0
Candidatos	Id	3	4	11	10	14	2	7	15	12	9	1
de 5	kbits/s	75	54	34	20	19	2	1	0	0	0	0
Candidatos	Id	15	12	1	9	5	3	14	11	10	4	2
de 7	kbits/s	195	151	58	1	1	1	0	0	0	0	0
Candidatos	Id	2	11	14	3	4	15	12	10	7	5	1
de 9	kbits/s	505	19	15	13	5	1	1	1	1	0	0
Candidatos	Id	3	11	5	14	2	4	9	15	12	7	1
de 10	kbits/s	22	22	20	15	10	2	1	0	0	0	0
Candidatos	Id	3	5	4	10	9	2	14	12	15	7	1
de 11	kbits/s	45	34	26	22	19	14	7	1	0	0	0
Candidatos	Id	7	1	15	11	9	4	14	10	5	3	2
de 12	kbits/s	151	113	20	1	1	1	0	0	0	0	0
Candidatos	Id	3	4	5	10	9	11	15	12	7	1	2
de 14	kbits/s	48	23	19	15	15	7	1	0	0	0	0
Candidatos	Id	1	7	12	14	9	11	10	5	4	3	2
de 15	kbits/s	404	195	20	1	1	0	0	0	0	0	0

Tabela 3.4: Filas de candidatos

Para cada passo a seguir, ilustraremos consecutivamente duas tabelas. A primeira representa cada *piconet* construída e a segunda tabela contém, para cada candidato a entrar na rede, os *Throughputs intra e extra piconet* no caso do candidato ser inserido

na *piconet* e a soma de quanto esse candidato deseja se comunicar com os dispositivos já inseridos na *piconet*.

Piconet 1	2							

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
9	505 kbits/s	101 kbits/s	505 kbits/s

Tabela 3.5: Exemplo de execução do algoritmo guloso (Passo 1.1)

Analisando a Tabela 3.5, verificamos que o dispositivo a ser inserido na *piconet* é o 9, já que é o único candidato a entrar na rede. Além disso, o *throughput* interno da *piconet* 1 não é extrapolado (505 kbits/s). Em seguida, removemos o 9 de todas as filas de candidatos e fazemos o primeiro dispositivo de sua fila de candidatos ser candidato a entrar na rede. A seguir, guardamos a posição desse dispositivo na *piconet* 1 (posição 2), pois os *throughputs intra e extra-piconet* passam a ser menores que os *throughputs* máximos (200 kbits/s *extra-piconet* e 923 kbits/s *intra-piconet*). Isto é necessário caso a rede não tenha suas necessidades de comunicação satisfeitas e, assim sendo, é necessário executar o algoritmo novamente com os novos valores de quebra.

Piconet 1	2	9						

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
11	538 kbits/s	203 kbits/s	19 kbits/s

Tabela 3.6: Exemplo de execução do algoritmo guloso (Passo 1.2)

Como o dispositivo 11 é candidato tanto do dispositivo 2 quanto do 9 e como o *throughput* interno da *piconet* 1 não é extrapolado (Tabela 3.6), este é inserido na *piconet* 1. Em seguida, removemos o 11 de todas as filas de candidatos e agora o primeiro dispositivo de sua fila de candidatos é seu candidato a entrar na rede.

Piconet 1	2	9	11					

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
4	581 kbits/s	261 kbits/s	31 kbits/s
14	560 kbits/s	287 kbits/s	22 kbits/s
3	603 kbits/s	305 kbits/s	58 kbits/s

Tabela 3.7: Exemplo de execução do algoritmo guloso (Passo 1.3)

Analisando a Tabela 3.7, verificamos que o candidato escolhido é o 3, pois o *throughput* interno da *piconet* 1 não é extrapolado e este é o candidato que mais deseja se comunicar com os dispositivos já inseridos na *piconet* 1 (58 kbits/s). Novamente removemos o 3 de todas as filas de candidatos e agora o primeiro dispositivo de sua fila de candidatos também é candidato a entrar na rede. O procedimento de remover o candidato escolhido de todas as filas de candidatos e fazer o primeiro dispositivo de sua fila de candidatos ser candidato a entrar na rede é repetido para cada dispositivo que é inserido na rede.

Piconet 1	2	9	11	3				

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
4	667 kbits/s	321 kbits/s	52 kbits/s
5	714 kbits/s	288 kbits/s	109 kbits/s
14	673 kbits/s	293 kbits/s	70 kbits/s

Tabela 3.8: Exemplo de execução do algoritmo guloso (Passo 1.4)

O próximo candidato a ser escolhido é o 5 (Tabela 3.8), pelas mesmas razões acima descritas.

Piconet 1	2	9	11	3	5			

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
4	832 kbits/s	196 kbits/s	106 kbits/s
14	803 kbits/s	238 kbits/s	89 kbits/s

Tabela 3.9: Exemplo de execução do algoritmo guloso (Passo 1.5)

Agora, o candidato escolhido é o 4 (Tabela 3.9). Além disso essa *piconet* termina pois todas as posições foram preenchidas (6 dispositivos mais 2 pontes). Então criamos a segunda *piconet* (*piconet* 2).

Piconet 1	2	9	11	3	5	4		
Piconet 2								

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	***Throughput desejado***
10	0 kbits/s	92 kbits/s	10 kbits/s com o 2
14	0 kbits/s	128 kbits/s	48 kbits/s com o 3
14	0 kbits/s	128 kbits/s	23 kbits/s com o 4
10	0 kbits/s	92 kbits/s	20 kbits/s com o 5
14	0 kbits/s	128 kbits/s	15 kbits/s com o 9
10	0 kbits/s	92 kbits/s	22 kbits/s com o 11

Tabela 3.10: Exemplo de execução do algoritmo guloso (Passo 1.6)

Para inserirmos o primeiro dispositivo da *piconet* 2, analisamos o *throughput* desejado dos candidatos, já que a *piconet* 2 ainda está vazia, e verificamos na Tabela 3.10 que o candidato que mais deseja se comunicar é o 14 (com o dispositivo 3).

Piconet 1	2	9	11	3	5	4		
Piconet 2	14							

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
10	15 kbits/s	190 kbits/s	15 kbits/s
15	1 kbits/s	747 kbits/s	1 kbits/s

Tabela 3.11: Exemplo de execução do algoritmo guloso (Passo 1.7)

Segundo a Tabela 3.11, o próximo candidato a ser escolhido é o 10, já que este é o dispositivo que mais deseja se comunicar com os dispositivos já inseridos na *piconet* 2. Além disso, guardamos a posição desse dispositivo na *piconet* 2 (posição 2), pois os *throughputs intra e extra-piconet* passam a ser menores que seus *throughputs* máximos (200 kbits/s *extra-piconet* e 834 kbits/s *intra-piconet*).

Piconet 1	2	9	11	3	5	4		
Piconet 2	14	10						

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
15	16 kbits/s	809 kbits/s	1 kbits/s
7	15 kbits/s	597 kbits/s	0 kbits/s
12	15 kbits/s	477 kbits/s	0 kbits/s

Tabela 3.12: Exemplo de execução do algoritmo guloso (Passo 1.8)

Agora, o candidato escolhido é o dispositivo 15 (Tabela 3.12).

Piconet 1	2	9	11	3	5	4		
Piconet 2	14	10	15					

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
12	36 kbits/s	1056 kbits/s	20 kbits/s
7	211 kbits/s	826 kbits/s	195 kbits/s
1	420 kbits/s	576 kbits/s	404 kbits/s

Tabela 3.13: Exemplo de execução do algoritmo guloso (Passo 1.9)

O próximo candidato escolhido é o dispositivo 1 (Tabela 3.13).

Piconet 1	2	9	11	3	5	4		
Piconet 2	14	10	15	1				

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
12	553 kbits/s	597 kbits/s	133 kbits/s
7	673 kbits/s	477 kbits/s	253 kbits/s

Tabela 3.14: Exemplo de execução do algoritmo guloso (Passo 1.10)

Agora o candidato escolhido é o dispositivo 7 (Tabela 3.14).

***Piconet* 1**	2	9	11	3	5	4		
***Piconet* 2**	14	10	15	1	7			

Candidatos	***T. Intra* de 2**	***T. Extra* de 2**	**Soma**
12	957 kbits/s	196 kbits/s	284 kbits/s

Tabela 3.15: Exemplo de execução do algoritmo guloso (Passo 1.11)

O candidato 12 é o último dispositivo inserido nesta *piconet* já que foi totalmente preenchida (Tabela 3.15).

***Piconet* 1**	2	9	11	3	5	4		
***Piconet* 2**	14	10	15	1	7	12		

Tabela 3.16: Exemplo de execução do algoritmo guloso (Passo 1.12)

Como não restam mais candidatos a serem inseridos na *scatternet* (Tabela 3.16), os dispositivos armazenados na fila de pontes são inseridos (como pontes) na rede como mostra a Tabela 3.17. Se depois de inseridas as pontes, restassem dispositivos na fila de pontes, esses dispositivos seriam inseridos em *piconets* no final da *scatternet*. Porém, não é o caso deste exemplo.

***Piconet* 1**	2	9	11	3	5	4	6	8
***Piconet* 2**	14	10	15	1	7	12	6	13
***Piconet* 3**	13	8						

Tabela 3.17: *Scatternet* resultante da primeira execução do algoritmo

A Tabela 3.18 apresenta os valores dos *Throughputs* desta *scatternet*.

	Throughput intra-piconet	***Throughput extra-piconet***
***Piconet* 1**	832 kbits/s	207 kbits/s
***Piconet* 2**	957 kbits/s	208 kbits/s
***Piconet* 3**	0 kbits/s	8 kbits/s

Tabela 3.18: Valores dos *throughputs* da primeira *scatternet* gerada

Desta forma, verificamos que a necessidade de comunicação desta rede não foi totalmente satisfeita, pois o *throughput* interno da *piconet* 1 extrapolou seu limite máximo em 15 kbits/s e o *throughput extra-piconet* em 7 kbits/s. O *throughput* interno da *piconet* 2 também extrapolou seu limite máximo em 141 kbits/s e o *throughput extra-piconet* em 8 kbits/s.

As próximas posições de quebra obtidas são: posição 2 para a *piconet* 1, posição 2 para a *piconet* 2 e posição 8 para a *piconet* 3 (*piconet* carregada).

Em seguida, percorremos todas as *piconets* verificando se a sua necessidade de comunicação foi satisfeita (**Procedimento Guloso**). Como já vimos, a *piconet* 1 não teve sua necessidade de comunicação satisfeita. Então, geramos a rede gulosa novamente com o novo valor de quebra igual a 2 para a *piconet* 1. Sendo assim, os novos valores de quebra para as *piconets* da *scatternet* são respectivamente 2, 8 e 8.

De forma similar a primeira execução, o algoritmo constrói novamente as filas de candidatos e a fila de dispositivos pontes. Verificamos que o primeiro dispositivo inserido é novamente o 2.

Piconet 1	2							

Candidatos	***T. Intra* de** 1	***T. Extra* de** 1	**Soma**
9	505 kbits/s	101 kbits/s	505 kbits/s

Tabela 3.19: Exemplo de execução do algoritmo guloso (Passo 2.1)

Analisando a Tabela 3.19, verificamos que o dispositivo a ser inserido na *piconet* é o 9, já que este é o único candidato a entrar na rede. Além disso, a condição de quebra é aceita, ou seja, a posição do dispositivo 9 na *piconet* é igual ao valor de quebra passado como parâmetro para o algoritmo guloso (posição 2) e os *throughputs intra* e *extra-piconet* não extrapolam seus limites máximos (923 kbits/s para o *throughput intra* e 200 kbits/s para o *extra-piconet*). Com isto, a *piconet* 1 termina e uma nova *piconet* é criada.

Piconet 1	2	9						
Piconet 2								

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	***Throughput desejado***
11	0 kbits/s	168 kbits/s	14 kbits/s com o 2
11	0 kbits/s	168 kbits/s	19 kbits/s com o 9

Tabela 3.20: Exemplo de execução do algoritmo guloso (Passo 2.2)

Como o dispositivo 11 é o único candidato (Tabela 3.20), este é o primeiro dispositivo a ser inserido na *piconet* 2.

Piconet 1	2	9						
Piconet 2	11							

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
4	26 kbits/s	260 kbits/s	26 kbits/s
17	7 kbits/s	282 kbits/s	7 kbits/s
3	45 kbits/s	310 kbits/s	45 kbits/s

Tabela 3.21: Exemplo de execução do algoritmo guloso (Passo 2.3)

O próximo candidato escolhido é o dispositivo 3 (Tabela 3.21).

Piconet 1	2	9						
Piconet 2	11	3						

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
4	92 kbits/s	360 kbits/s	47 kbits/s
5	154 kbits/s	297 kbits/s	109 kbits/s
14	100 kbits/s	328 kbits/s	55 kbits/s

Tabela 3.22: Exemplo de execução do algoritmo guloso (Passo 2.4)

Agora, o candidato escolhido é o dispositivo 5 (Tabela 3.22).

Piconet 1	2	9						
Piconet 2	11	3	5					

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
4	255 kbits/s	239 kbits/s	101 kbits/s
14	228 kbits/s	277 kbits/s	74 kbits/s

Tabela 3.23: Exemplo de execução do algoritmo guloso (Passo 2.5)

O próximo candidato escolhido é o dispositivo 4 (Tabela 3.23).

Piconet 1	2	9						
Piconet 2	11	3	5	4				

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
10	321 kbits/s	199 kbits/s	66 kbits/s
14	352 kbits/s	173 kbits/s	97 kbits/s

Tabela 3.24: Exemplo de execução do algoritmo guloso (Passo 2.6)

Agora o candidato escolhido é o dispositivo 14 (Tabela 3.24).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14			

Candidatos	***T. Intra* de** 2	***T. Extra* de** 2	**Soma**
10	433 kbits/s	103 kbits/s	81 kbits/s
15	353 kbits/s	792 kbits/s	1 kbit/s

Tabela 3.25: Exemplo de execução do algoritmo guloso (Passo 2.7)

O dispositivo 10 é o último candidato desta *piconet* a ser escolhido (Tabela 3.25), pois a *piconet* está totalmente carregada. Então criamos a terceira *piconet* (*piconet* 3).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14	10		
Piconet 3								

Candidatos	***T. Intra* de** 3	***T. Extra* de** 3	***Throughput desejado***
15	0 kbits/s	621 kbits/s	0 kbits/s com o 2
7	0 kbits/s	407 kbits/s	1 kbit/s com o 3
12	0 kbits/s	287 kbits/s	1 kbit/s com o 4
7	0 kbits/s	407 kbits/s	1 kbit/s com o 5
15	0 kbits/s	621 kbits/s	1 kbit/s com o 9
15	0 kbits/s	621 kbits/s	0 kbits/s com o 10
12	0 kbits/s	287 kbits/s	1 kbit/s com o 11
15	0 kbits/s	621 kbits/s	1 kbit/s com o 14

Tabela 3.26: Exemplo de execução do algoritmo guloso (Passo 2.8)

Analisando a Tabela 3.26, verificamos que o primeiro candidato a ser inserido na *piconet* 3 é o dispositivo 7, já que este foi o primeiro a possuir o maior *throughput* desejado (1 kbit/s).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14	10		
Piconet 3	7							

Candidatos	***T. Intra* de** 3	***T. Extra* de** 3	**Soma**
15	195 kbits/s	638 kbits/s	195 kbits/s
12	151 kbits/s	392 kbits/s	151 kbits/s

Tabela 3.27: Exemplo de execução do algoritmo guloso (Passo 2.9)

O próximo candidato escolhido é o dispositivo 15 (Tabela 3.27).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14	10		
Piconet 3	7	15						

Candidatos	***T. Intra* de** 3	***T. Extra* de** 3	**Soma**
12	366 kbits/s	583 kbits/s	171 kbits/s
1	657 kbits/s	289 kbits/s	462 kbits/s

Tabela 3.28: Exemplo de execução do algoritmo guloso (Passo 2.10)

Agora o candidato escolhido é o dispositivo 1 (Tabela 3.28).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14	10		
Piconet 3	7	15	1					

Candidatos	***T. Intra* de** 3	***T. Extra* de** 3	**Soma**
12	941 kbits/s	8 kbits/s	284 kbits/s

Tabela 3.29: Exemplo de execução do algoritmo guloso (Passo 2.11)

O último candidato inserido na *scatternet* é o dispositivo 12, pois não restam mais candidatos a entrar na rede (Tabela 3.29).

Piconet 1	2	9						
Piconet 2	11	3	5	4	14	10		
Piconet 3	7	15	1	12				

Tabela 3.30: Exemplo de execução do algoritmo guloso (Passo 2.12)

Depois de inserido o último candidato (Tabela 3.30), os dispositivos pontes são inseridos na rede, como mostra a Tabela 3.31.

Piconet 1	2	9	6	8				
Piconet 2	11	3	5	4	14	10	6	13
Piconet 3	7	15	1	12	13	8		

Tabela 3.31: *Scatternet* resultante da segunda execução do algoritmo

A Tabela 3.32 apresenta os valores dos *Throughputs* desta *scatternet* gerada.

	Throughput intra-piconet	***Throughput extra-piconet***
***Piconet* 1**	505 kbits/s	112 kbits/s
***Piconet* 2**	433 kbits/s	115 kbits/s
***Piconet* 3**	941 kbits/s	16 kbits/s

Tabela 3.32: Valores dos *throughputs* da segunda *scatternet* gerada

Como a necessidade de comunicação desta *scatternet* foi totalmente satisfeita, já que nenhum dos *throughputs* extrapolou o limite máximo, o algoritmo termina aqui.

Este exemplo não utilizou o método de *BackTracking*, pois sua necessidade de comunicação foi satisfeita sem que fosse necessário percorrer todas as possíveis quebras.

É importante observar que a *scatternet* resultante é exatamente a mesma construída no exemplo do gerador de redes aleatórias Bluetooth (ver Figura 3.2). Porém isto nem sempre ocorre pois, às vezes, um problema de satisfação como este, pode ter mais de uma solução.

Capítulo 4

Resultados Obtidos

Os testes com o algoritmo guloso foram feitos num computador com processador Intel Pentium II 300, com 192 Mbytes de RAM, usando sistema operacional Windows 98 e os algoritmos foram implementados na linguagem C. Testamos a eficiência do algoritmo na configuração e reconfiguração de *scatternets* Bluetooth utilizando diferentes topologias de rede.

Os dados foram calculados para um tempo máximo de execução do algoritmo de 1,5 segundos, que pode ser considerado um tempo razoável para diversos tipos de aplicações Bluetooth. Fixando o tempo de execução em 1,5 segundos, deixamos de satisfazer apenas 0,46% das 24.000 redes testadas.

4.1 Configuração

Na configuração, supomos que a necessidade de comunicação entre os dispositivos da rede é previamente conhecida. Para isso, utilizamos o nosso gerador de redes aleatórias Bluetooth para construir redes com as topologias Anel e Aleatória. Para cada uma delas geramos 4.000 redes, sendo 500 para cada tamanho de rede em número de *piconets*, de 3 a 10.

Primeiramente, geramos as redes em anel e utilizamos o algoritmo guloso para resolvê-las também em anel. Os resultados obtidos são apresentados na Tabela 4.1. Analisando esses dados, verificamos que o resultado do algoritmo foi excelente, já que ele conseguiu satisfazer quase 99,5% de todas as redes testadas. O tempo médio gasto pelo algoritmo

para tentar satisfazer as redes geradas e resolvidas em anel foi de 0,02 segundos.

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Redes Satisfeitas	100,0	100,0	99,8	99,8	99,4	99,0	99,0	98,8	**99,48**

Tabela 4.1: Resultados obtidos para redes geradas e resolvidas em Anel

Em seguida, geramos as redes com uma topologia aleatória e novamente resolvemos em anel, para verificar se a eficiência do algoritmo continua a mesma. Os resultados obtidos são apresentados na Tabela 4.2.

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Redes Satisfeitas	94,2	93,8	88,6	82,8	73,6	68,4	54,8	49,8	**75,75**

Tabela 4.2: Resultados obtidos para redes geradas Aleatoriamente e resolvidas em Anel

Esta Tabela mostra que houve uma forte queda na eficiência do algoritmo, pois ele conseguiu satisfazer apenas 75,75% das redes. A principal queda ocorreu nas redes geradas com um grande número de *piconets* (menos de 50% das redes geradas com 10 *piconets* foram satisfeitas). Neste caso, o tempo médio gasto pelo algoritmo foi de 0,14 segundos.

Para tentar melhorar o índice de satisfação para redes geradas com uma topologia aleatória, construímos a topologia de rede que chamamos de Triângulo-Aresta. Essa topologia foi utilizada, pois sua estrutura assemelha-se com a idéia gulosa do algoritmo, conectando as *piconets* que mais precisam se comunicar (*piconets* geradas consecutivamente) de forma não-linear. Ela também respeita o limite máximo de três pontes por *piconet* para não degradar o desempenho interno da *piconet*. Os resultados obtidos para as redes geradas com uma topologia aleatória e resolvidas em triângulo-aresta, são apresentados na Tabela 4.3.

Analisando esta Tabela, verificamos que houve uma melhora no índice de satisfação, que no total aumentou de 75,75% para 83,13%. O aumento mais acentuado ocorreu nas redes geradas com mais de 7 *piconets*. O tempo médio gasto pelo algoritmo para tentar

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Redes Satisfeitas	98,2	92,6	88,0	88,0	80,0	76,8	73,6	67,8	**83,13**

Tabela 4.3: Resultados obtidos para redes geradas Aleatoriamente e resolvidas em Triângulo-Aresta

satisfazer as redes geradas aleatoriamente e resolvidas em triângulo-aresta foi de 0,13 segundos.

Finalmente, para um efeito comparativo com a Tabela 4.5 da próxima seção, geramos e resolvemos redes com a topologia Triângulo-Aresta. Os resultados obtidos são apresentados na Tabela 4.4.

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Redes Satisfeitas	100,0	96,4	95,0	95,2	91,6	89,2	82,8	81,8	**91,5**

Tabela 4.4: Resultados obtidos para redes geradas e resolvidas em Triângulo-Aresta

Esta Tabela mostra que o algoritmo obteve um bom resultado, satisfazendo 91,5% das redes testadas. Neste caso, o tempo médio gasto pelo algoritmo foi de 0,08 segundos.

Com os dados apresentados acima, verificamos que os melhores índices foram obtidos para redes geradas e resolvidas com a topologia anel (99,48%). No entanto, não é muito comum encontrarmos cenários Bluetooth que possuam uma necessidade de comunicação caracterizada por uma topologia anel. O mais comum é encontrarmos cenários onde as redes possuem uma necessidade de comunicação aleatória caracterizada também por uma topologia aleatória. Por este motivo, geramos as redes com a topologia aleatória. Entretanto, quando tentamos resolvê-las utilizando a topologia anel, verificamos que o índice de satisfação do algoritmo foi reduzido de 99,48% para 75,75%. Para tentar melhorar este índice, utilizamos a topologia triângulo-aresta, a qual se assemelha com a idéia gulosa do algoritmo, para resolvê-las. Com isso, o índice aumentou para 83,13%. Considerando que as redes possuem uma alta taxa de comunicação, apesar de terem sido construídas com uma necessidade de comunicação aleatória, um índice de satisfação de quase 85% pode

ser considerado bom.

4.2 Reconfiguração

Para simular uma reconfiguração, construímos as redes através do nosso gerador (redes factíveis) e, em seguida, aumentamos o valor da necessidade de comunicação de apenas um enlace que conecta quaisquer dois dispositivos da rede, de forma a extrapolar o *throughput intra-piconet* ou *extra-piconet* máximo. A intenção aqui é simular uma rede já configurada, que depois de um certo tempo, passa a não suportar a necessidade de comunicação. É importante ressaltar que quando ocorre essa mudança na rede, não sabemos se essa nova rede é factível (possui pelo menos uma solução), pois a rede antiga, apesar de ser construída aleatoriamente, possui uma alta taxa de comunicação.

Para um efeito comparativo ao nosso algoritmo guloso, construímos um algoritmo de busca local, que percorre toda a rede, verificando se a troca entre quaisquer dois dispositivos não-pontes, pertencentes a *piconets* diferentes, melhora o desempenho da rede. Porém somente a melhor troca é aceita (realizada). Em seguida, percorremos novamente a rede, realizando o mesmo procedimento. Isso é feito até que mais nenhuma troca seja realizada ou quando a necessidade de comunicação da rede seja satisfeita. Esse algoritmo foi escolhido, pois ele utiliza a estrutura (topologia) da rede atual como ponto de partida para tentar resolver o problema, enquanto o nosso algoritmo cria uma rede totalmente nova sem saber a topologia antiga.

Na reconfiguração, geramos o mesmo número de redes da configuração, entretanto utilizamos as topologias Aleatória e Triângulo-Aresta, pois esta última foi a que obteve o melhor resultado na configuração de redes geradas com uma topologia aleatória. O tempo de execução máximo foi mantido em 1,5 segundos.

Os resultados obtidos para as redes geradas e reconfiguradas em triângulo-aresta são apresentados na Tabela 4.5.

Analisando os dados desta Tabela, verificamos que para a topologia triângulo-aresta, o algoritmo guloso foi bem mais eficiente (39,93%) que o algoritmo de busca local (12,78%), já que ele conseguiu satisfazer mais que o triplo do número de redes satisfeitas pela busca local. É importante lembrar que esses baixos índices de satisfação (menores que 41%)

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Satisfeitas Busca Local	8,0	9,0	9,0	21,0	9,8	16,8	17,4	11,2	**12,78**
% Satisfeitas Guloso	33,2	36,2	42,2	42,6	43,2	41,6	40,2	40,2	**39,93**

Tabela 4.5: Resultados obtidos para redes geradas e reconfiguradas em Triângulo-Aresta

ocorrem porque as redes já estão saturadas, tendo em vista que na configuração, para redes geradas e resolvidas nessa topologia, o índice total de satisfação foi de 91,5% (Tabela 4.4 da seção anterior). Além disso, não sabemos se a rede resultante é factível. O tempo médio gasto pelo algoritmo guloso para tentar satisfazer as redes geradas e reconfiguradas em triângulo-aresta foi de 0,24 segundos. Já o tempo médio gasto algoritmo de busca local foi de 0,21 segundos.

Para as redes geradas com uma topologia aleatória e reconfiguradas em triângulo-aresta, os resultados obtidos estão apresentados na Tabela 4.6.

Nº *Piconets*	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Nº Redes Geradas	500	500	500	500	500	500	500	500	**4000**
% Satisfeitas Busca Local	8,0	12,4	17,2	8,8	13,6	9,8	13,8	10,4	**11,75**
% Satisfeitas Guloso	36,0	40,4	38,6	35,2	38,8	37,2	35,8	32,2	**36,78**

Tabela 4.6: Resultados obtidos para redes geradas Aleatoriamente e reconfiguradas em Triângulo-Aresta

Esta Tabela mostra que para redes geradas com uma topologia aleatória e reconfiguradas em triângulo-aresta, o algoritmo guloso novamente foi bem mais eficiente (36,78%) que o algoritmo de busca local (11,75%), satisfazendo mais que o triplo do número de redes satisfeitas pela busca local. Neste caso, o tempo médio gasto pelo algoritmo guloso foi de 0,25 segundos. Já o tempo médio gasto algoritmo de busca local foi de 0,21 segundos.

Analisando os dados apresentados, verificamos que o algoritmo guloso foi muito mais eficiente que o algoritmo de busca local, tanto para as redes que possuíam uma topologia triângulo-aresta quanto para as que possuíam uma topologia aleatória. Os dados também mostraram que o algoritmo guloso foi mais eficiente na resolução das redes que já possuíam uma topologia triângulo-aresta.

Capítulo 5

Conclusão

Este trabalho apresenta um algoritmo guloso para configurar e reconfigurar *scatternets* Bluetooth de acordo com as necessidades de comunicação entre os dispositivos, na tentativa de satisfazê-las. Esta é uma nova abordagem para o problema da formação de *scatternets* Bluetooth, já que nenhum dos trabalhos encontrados que abordam este assunto levam em consideração as necessidades de comunicação entre os dispositivos da rede.

Além disso, construímos um gerador de redes aleatórias Bluetooth, que para uma dada topologia de rede (anel, triângulo-aresta e aleatória), este gera tráfego aleatório entre os dispositivos de forma a não extrapolar os limites máximos de comunicação desta rede.

É importante observar que a idéia gulosa, utilizada pelo algoritmo (guloso), é muito eficiente na resolução desse tipo de problema, colocando os dispositivos que se comunicam muito numa mesma *piconet*, já que o canal *intra-piconet* possui o maior *throughput*, ou em *piconets* próximas, gerando assim, menos tráfego entre *piconets*.

Os resultados mostraram que o algoritmo foi muito eficiente na configuração de redes geradas e resolvidas em anel (99,48%) e que a melhor topologia a ser utilizada na configuração de redes geradas com uma topologia aleatória é a triângulo-aresta (83,13%), pois sua estrutura assemelha-se com a idéia gulosa do algoritmo. Na reconfiguração, tanto para redes geradas e reconfiguradas em triângulo-aresta, quanto para redes geradas com uma topologia aleatória e reconfiguradas em triângulo-aresta, o algoritmo guloso foi três vezes mais eficiente que o algoritmo de busca local, construído para efeito comparativo.

Como trabalhos futuros, pretendemos testar o algoritmo guloso inserindo cenários onde

uma ponte pode estar conectada a mais de duas *piconets* e investigar novas topologias de rede que melhorem ainda mais o índice de satisfação das *scatternets* Bluetooth. Além disso, pretendemos estudar algumas políticas para a escolha do dispositivo mestre de cada *piconet* da *scatternet*, pois o mestre controla o tráfego interno da *piconet*.

Referências Bibliográficas

[1] Jaap Haartsen. Bluetooth-the universal radio interface for ad hoc, wireless connectivity. *Ericsson Review*, pp. 110–117, 1998.

[2] Jaap Haartsen. The bluetooth radio system. *IEEE Personal Communication*, pp. 28–36, Fevereiro 2000.

[3] Jaap Haartsen, Mahmoud Naghshineh, e Jon Inouye. Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, pp. 1–8, 1998.

[4] P. Johanson, Y. Lee, M. Gerla, e M. Kazantzidis. Bluetooth - an enabler for personal area networking. *IEEE Network, Special Issue in Personal Area Networks*, Setembro-Outubro 2001.

[5] Wanderley Ravagnani Junior, Edmundo R. M. Madeira, e Jacques Wainer. A greedy algorithm to configure and reconfigure bluetooth scatternets. *The Fifth IEEE Conference on Mobile and Wireless Communications Networks (IEEE MWCN'2003)*, Cingapura, Outubro 2003.

[6] Wanderley Ravagnani Junior, Edmundo R. M. Madeira, e Jacques Wainer. Um algoritmo guloso para configuração e reconfiguração de scatternets bluetooth. *Seminário Integrado de Software e Hardware (SEMISH) - SBC*, Campinas, Agosto 2003.

[7] Wanderley Ravagnani Junior, Jacques Wainer, e Edmundo R. M. Madeira. Um algoritmo para configuração de scatternets bluetooth. *IV Workshop de Comunicação sem Fio e Computação Móvel*, pp. 184–190, São Paulo, Outubro 2002.

[8] Manthos Kazantzidis e Mario Gerla. On the impact of inter-piconet scheduling in bluetooth scatternets. *International Conference on Internet Computing*, pp. 37–43, 2002.

[9] Ching Law, Amar K. Mehta, e Kai-Yeung Siu. Perfomance of a new bluetooth scatternet formation protocol. *ACM Symposium on Mobile ad Hoc Networking and Computing*, Outubro 2001.

[10] Werner Priess, José Ferreira de Rezende, Luiz Fernando Rust da C. Carmo, e Luci Pirmez. Bluenets: Uma ferramenta para estudo do desempenho do bluetooth. *IV Workshop de Comunicação sem Fio e Computação Móvel*, pp. 161–171, São Paulo, Outubro 2002.

[11] Lakshmi Ramachandran, Manika Kapoor, e Alok Aggarwal. Clustering algorithms for wireless ad hoc networks. *SIG Mobile*, 2001.

[12] András Rácz, György Miklós, Ferenc Kubinzky, e András Valkó. A pseudo random coordinated scheduling algorithm for bluetooth scatternets. *ACM Symposium on Mobile ad Hoc Networking and Computing*, Outubro 2001.

[13] Theodoros Salonidis, Pravin Bhagwat, Leandros Tassiulas, e Richard LaMaire. Proximity awareness and ad hoc establishment in bluetooth. *Infocom*, 2001.

[14] Rob Shepherd. Bluetooth wireless technology in the home. *Eletronics and Communication Engineering Journal*, pp. 195–203, Outubro 2001.

[15] SIG. *Bluetooth Core Specification, version 1.1*, Maio 2002. http://www.bluetooth.com.

[16] Godfrey Tan. Self-organizing bluetooth scatternets. Tese de Mestrado, Massachusetts Institute of Technology, Janeiro 2002.

[17] Godfrey Tan e John Guttag. A locally coordinated scatternet scheduling algorithm. *IEEE Conference on Local Computer Networks (LCN)*, Novembro 2002.

[18] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Quarta Edição, 2003.

[19] Sergio Gonzalez Valenzuela, Son T. Vuong, e Victor C. M. Leung. Efficient formation of dynamic bluetooth scatternet via mobile agent processing. *5th International Workshop on Mobile Agents for Telecommunication Applications - MATA'03*, pp. 1–10, Marrakech, Marrocos, LNCS n. 2881, Springer-Verlag, Outubro 2003.