平成２２年度　特別研究報告書

MANETによる店舗情報通知の改善

龍谷大学　理工学部　情報メディア学科
学籍番号　　T060590　辻川　将成
指導教員　　三好　力　教授

内容梗概

近年、インターネット上において多くの店舗情報を調べるサイトがある。場所を調べたり、値段を他の店と比較したりする事はよくあることである。しかし、出かけ先で特に入りたい店も決まっていないが、適当な店に入りたい事がよくある。現在位置を検索するとある程度の情報は出てくるが、店舗の情報だけが出て来ないので検索し辛い。
そこで、アドホックネットワークを用いて、現在位置周辺の店舗情報のみ表示するシステムの構築を試みる。また、位置情報の履歴を元に、進行方向を予測し数分先の位置を推定することで、店舗情報数が減少するまでの時間を長くする事を試みた。

目次

第一章 はじめに

1.1. 本研究の背景

近年、携帯電話や携帯ゲーム機をはじめとした携帯端末の普及率はとても高いものとなっている。急速な技術の向上が見られ、それに伴って、無線通信技術も向上した。それらの端末を使用してインターネットにアクセスする事はとても容易で、一般的になった。長距離無線通信を利用してのインターネットは代表格であるが、近距離無線通信も利用されている。例えばニンテンドー DS や PSP には無線通信によりインターネットを介してオンラインでプレイが出来るが、更に端末同士で直接接続するアドホックモードを搭載している。

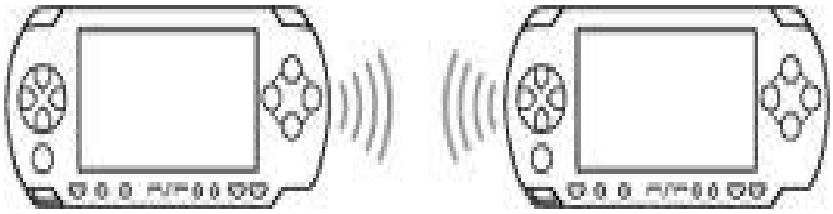

図 1

アドホックネットワークとは、無線基地局等のインフラストラクチャーを必要としないネットワークの事である。そのため、インフラストラクチャーを導入するコストや導入するまでの時間を削減することが可能となる。その中でも、モバイルアドホックネットワーク（MobileAd-hocNETwork＝MANET）は、Mobile という名の通り、動的なトポロジー変化をし、ノードは移動することになる。
現在、携帯電話は機能が多様化しており、GOS 機能を内蔵したものも多い。インターネットを通じて、お店の紹介サイトから情報やクーポンを得る人も多い。

図 2

携帯端末において、近距離無線を用いる事は、特定の地域に固有の情報や、時間を限定した情報を共有することを可能にする。

第二章　アドホックネットワーク

アドホックネットワークは、インフラストラクチャーを必要としないネットワーク体系であり、一対一の通信である。インフラストラクチャーを持たない場所で簡単に、安価でネットワークを構築することが出来るのが最大の利点。災害時、通信設備が整っていない環境でも通信を行える事に注目が集まっている。他にも車々間通信、センサネットワークにも研究が盛んである。
宛先と送信ノードが固定されず、どのノードも宛先、送信ノードになれる。多数の端末をアクセスポイント無しに相互に接続する形態＝マルチホップ通信が可能である。

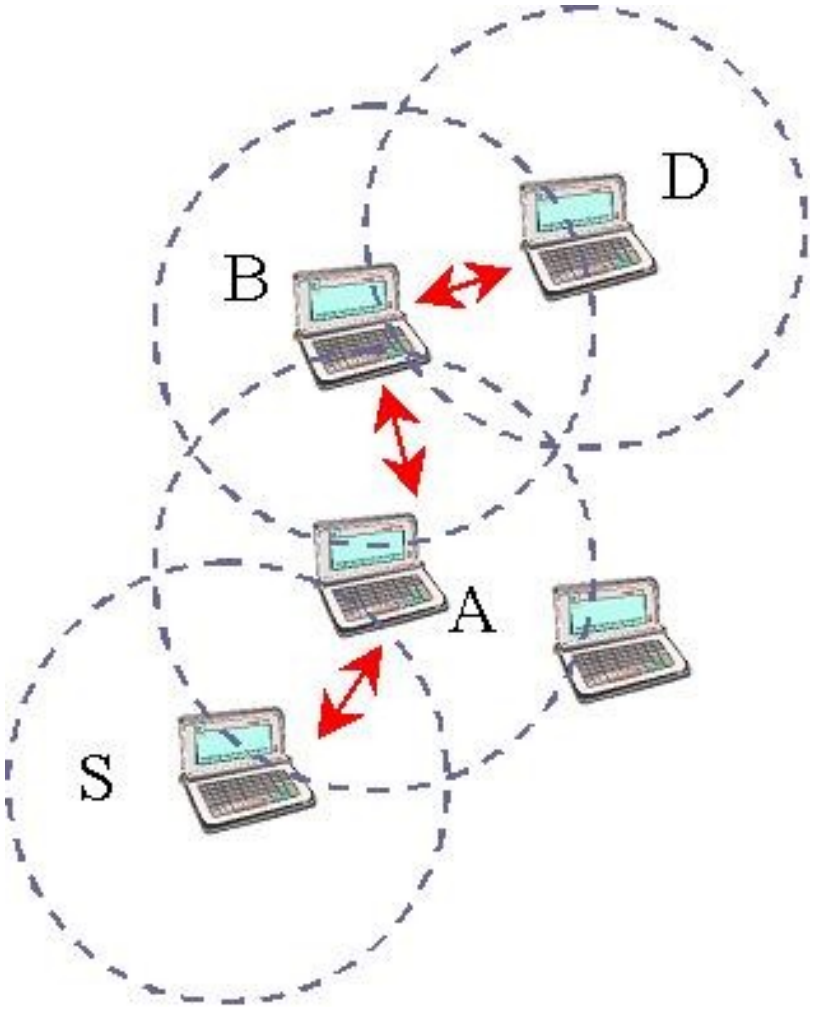

図 3

モバイルアドホックネットワーク(MANET)では、ノード同士は対等な関係であり、動的に変化する。通信経路をどのように決定するのかが問題になる。

2.1 ルーティング

ルーティングとは、経路制御のことであり、ルーティングテーブルに沿ってパケットデータを送信元ノードから宛先ノードまで転送するものである。ルーティングテーブルは経路表ともいい、各ノードに記録されていて、宛先とその宛先へパケットを届けるために送り出すべき転送先の組が表になったものである。コンピュータネットワーク OSI 基本参照モデル

2.2 ルーティングプロトコル

ノード同士は自由に動き回るため、ルーティングテーブルは絶えず変化しなければならない。その時にルーティングテーブルを自動で取り決める仕組みが必要になる。それがルーティングプロトコルである。マルチホップ無線ネットワークではノードが頻繁に移動していて、ノードの接続が繋がったり切れたりを繰り返している。様々な環境により、最適のルーティングプロトコルは違ってくる。
色々なルーティングプロトコルがあるが、リアクティブ型とプロアクティブ型の 2 つに分けられる。その両方の特徴を合わせ持つものがハイブリッド型である。

2.2.1・リアクティブ型

リアクティブ型プロトコルは、通信要求があった場合にルーティングプロトコルが動作し、通信要求がない場合は経路制御パケットは送信されない。よって、経路表も通信要求が無い限りいつまでも造られない。通信要求を出すと、周りのノードの存在を探し、経路表を作っていく。しかし、複数のノードから短時間のうちに通信要求があると、大量のデータパケットと経路制御パケットでネットワークに負担がかかる。また、ノードを探すために電波を発信していると、すぐに端末の電池が無くなってしまう。各ノードは移動しており、経路表はすぐに意味のないものになってしまう。
そのため、通信が起こる直前に必要に応じて経路表を作る。また、経路探索によって作られた通信経路を再利用することによって負荷を減らしている。リアクティブ型プロトコルは、経路情報が多く各ノードの移動が頻繁に起こるネットワークに適している。

2.2.2・プロアクティブ型

プロアクティブ型プロトコルは、事前に通信経路を確立し、各ノードにおいて定期的に経路情報を交換する。常に経路情報を持つので、ノードの移動による経路の変更が大きいと、再構築する手間がかかり望ましくない。だが、事前に通信経路を確立しているため、ノードの通信リクエストがあればすぐに対応が出来る。したがって、ノードの移動が少ないネットワークにおいては適している。

2.3　フラッディング

経路情報を構築するのではなく、経路発見のための経路制御パケットを送信するもので、経路を発見していく。送信元ノードは通信が可能なノードに対しデータコピーを送信し、データを受け取ったノードも他のノードにデータコピーを送信する。それを繰り返すことでデータが宛先に送信される。

2.4　GPS

グローバルポジショニングシステム

図 4

衛星によって緯度経度情報を元に現在位置を測定するものであり、我々の生活の中では主にカーナビゲーションシステムで利用されている。携帯電話では精度によって様々だが、良いものならば３０Ｍ以内の精度で求めることが出来る。

2.5　広告の配信について

世の中のな色んな所で見かける広告。その種類には様々なものがある。同じようなものは一つと大分すると

・テレビやラジオ等の CM
・新聞やチラシ、雑誌等のペーパー
・看板
・電車の吊り広告
・街頭での広告配布
・web 上でのバナーや検索結果で表示
・ダイレクトメール
・メール広告

等が我々の生活に密接したものだろう。
中でも、街頭での広告配布について焦点を当てる。
この広告は近くの店舗の情報であり、その日に配られた情報なので最新の情報を手に入れることもできる。

しかしこの方法を用いる時、問題は以下のように考えられる。

・配布自体に人を使うため、人件費がかかる。
・配布場所がとても限定される。
・１体１の手渡しの為、情報伝達効率が低い。

・受け取ってもらえる可能性が低い
・配布するものに宣伝効率を上げるために、ポケットティッシュ等付加価値を与える多く、コストが更に上がる場合がある。
・受け取ってもらえても、広告を見ない可能性が高い。
・広告のポイ捨て。
・一方的に情報が伝達されるため、本当に安いものか、良いものなのか他の店と比較がしにくい。

上記の問題点はアナログでの広告にはどうしても回避しにくい。

デジタルでの宣伝について考える。
ネットワークを利用し、電子的に広告を配布する手段として挙げられるものが、インターネットがある。情報の量がたくさんあるため、沢山の情報を入手する事が可能である。

・インフラストラクチャーの構築に莫大なコストがかかる。
・全国の情報が見れるが、地域性に劣る。
・情報の量がありすぎて、検索の仕方によっては目的の情報が手に入れられない。本当に情報が無いのかがわからない。様々なサイトに登録されて似たような情報が多く、時間の無駄になることが多々ある。
・高齢者等、パソコンが使えない人には全く情報が見えない。

2.6　アドホックネットワーク利用の店舗情報受け取りシステム

インターネット広告はインフラストラクチャーに左右されるという点があるが、それとは異なり、電子的だがインフラストラクチャーに影響されない手段としてアドホックネットワークがある。
しかし新たな問題も出てくる。

・基本的に携帯端末で行うため、消費電力の問題。
・広告を希望するノード以外にも広告が配信され、余分な情報の受信が多くなる。
・人から人へと広告が配信されるため、広告の配信量や配信効果を知ることが難しい。そのため、手を出しづらい。
・アドホックネットワークの利用者が少ないと、あまり効果が出ない
・プライバシーやセキュリティ等、アドホックネットワーク特有の弱点が付きまとう。
・チェーンメールやダイレクトメールのような、不必要な情報の氾濫。

そこで、問題に対して考えれる要件としては
・端末エネルギー消費の削減
・広告配信の際、情報の選定をする
・セキュリティ性の向上
というものが挙げられる。

中でも、広告配信際に情報の選定をするという事に焦点を挙げていきたい。

店舗情報を取得する既存技術

既存の、店舗情報を取得する方法として、NTT ドコモがサービスの提供をしている、i コンシェルというものがある。
これは登録店舗の情報が更新されると通知してくれたり、イベントの情報が日にちごとにわかるものである。月額料金が発生し、2011/1/18 現在、契約店舗数 は 14,603 店ある。

第三章　提案手法

3.1　問題点

従来考案されていた店舗情報受け取りシステムは情報発信を行っている店舗を通過したときにその店の店舗情報を受け取る。近くの店舗が情報発信していても、そこを通らないと情報を得られないため、使いどころが限定される。さらに、自宅等の離れている場所からでもインターネットを通じて店舗情報の検索が出来るので、利点がかなり少なくなってしまう。

そこで
1）離れている場所からでも通信
2）多数の店舗情報を受信
3）インターネットとは違った利点
という３点が要点となってくる。

3.2　提案手法

本研究では、店舗以外の場所でその店舗情報を受け取り、さらに行き先を予測して店舗情報を受け取るかどうかを決めることを考える。
提案手法でも従来手法と同じく通過した店舗の店舗情報を収集する。さらに、離れた場所の店舗情報を得るため、同様の端末とアドホック通信が可能になるたびに店舗情報を交換する。その際、送信時は現在位置から近い順に送信し、受信時はデータ数が一定値を超えると、現在位置から遠い順に店舗情報を消去する事で近くの店舗情報を優先的に収集する。これにより、広範囲の店舗情報を得ることができる。

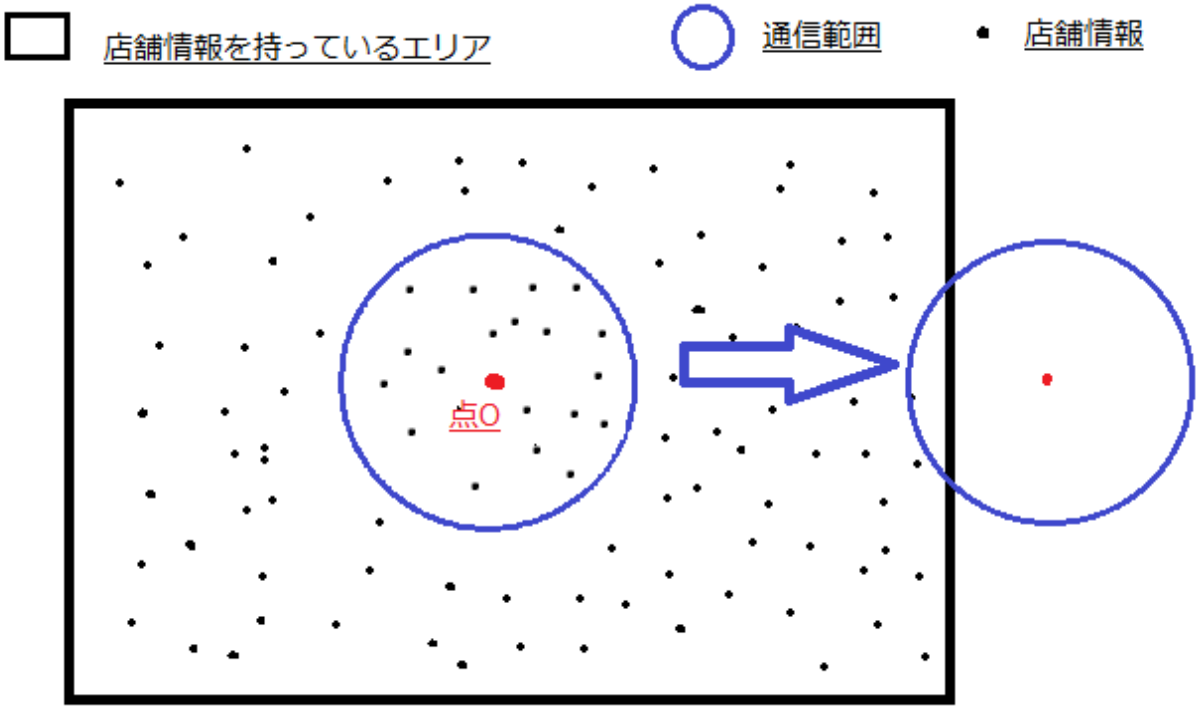

図 5

図 5 のようにエリア内には無数の店舗が存在し、エリア内の店舗情報をランダムに持っているとする。点 O が移動したとき、現在位置が変化すると店舗情報数は絶えず変化する。その時、エリアの外に近づくに従い、店舗情報数は少なくなっていく。この特性を使って手法の有効性を検証する。

他者から店舗情報を受け取る場合、その店舗情報は相手の来た位置に依存することになる。自分の進行方向から来た相手はその進行方向の情報を持っていると考えられ、情報を交換するのに望ましい。また、進行方向から多く情報を貰う事で、店舗情報を持っているエリアから出にくく出来る。　従って、相手の来た方向を特定し、自分の行きたい方向から来た相手のみと店舗情報を交換する事が必要となってくる。これを提案手法 2 とする。

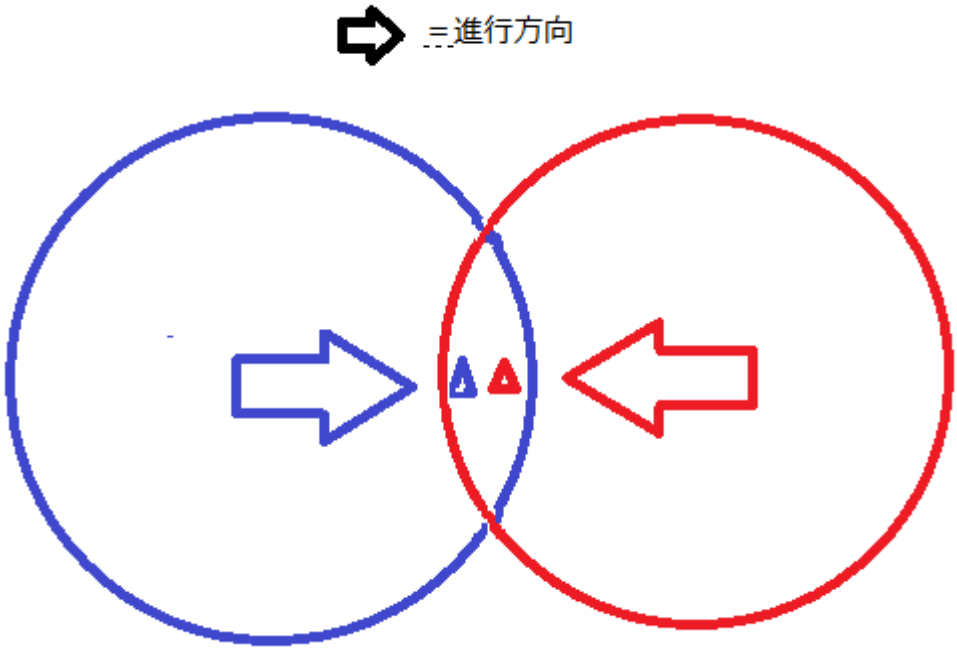

図 6

実際に歩く場合は角で曲がったりと不規則な場合が多いので、その位置履歴を記録し、行き先を決定するための歩行シミュレーターを作成する。その結果から進行方向を決定するのに十分であるかを判断する。

図 7

第四章　実験方法

4.1　実験概要

エリア内に位置情報を持った店舗を 500 個設置する。
シミュレーションで実験を行い、起点をずらした時のノード数の変化を見ることで、行き先を推定する事が有効となるかの検討を行う。

4.2　実験方法

本実験では、ランダムに打たれた点が設定範囲内にある数を調べる。その設定範囲が移動するときの数を記録していき、その比較評価するため、C 言語でのシミュレーションを行った。

アルゴリズム

1.ノード数を数えるプログラム

・ノード(店舗情報)の数、ノードが存在するエリアの範囲、通信半径、原点の座標の設定
・全てのノードをエリア内にランダムに配置する
・ノード数を記録する
・原点を X 軸方向にずらしていき、(通信半径＋ノードが存在するエリアの範囲)を超えるまでずらす
～これを 100 回繰り返す

2.歩行シミュレーターのプログラム
・x 方向のブロック個数を決める。奇数になるようにする。
・Y 方向のブロック個数を決める。奇数になるようにする。
・ブロックサイズを決める。
・現在位置、移動の向き、１つ前の位置を記録する。
・歩行の速さを決める。
・バックグラウンドの黒枠のサイズを決める。
・全体をクリア
・外枠をセット
・基準点をセット
・マップ作成。
・乱数で、壁を触るのが右か左かを決める。
・描画領域を黒にする。
・壁を白で表示。
・外枠を描く。
・探索範囲と今までの最短経路を表示。
・直前の位置を深緑で、現在の位置を緑で表示。
・直前の進行方向を黒で、現在の進行方向を赤で表示。
・ブロックの升目を fill カラーで塗り、内側に line カラーで矩形を追加。
・進行方向を小さい矩形で表示。後戻りが生じていれば、フラグを消す。
・ゴールに着いたら最終経路を青と黒線で表示。
・ループし、次の処理を繰り返す。

実験環境
実験環境は、1000×1000 の空間において 500 個、50000 個のノードをランダムに配置させる。通信半径は 100、250 とする。
パラメータ毎に 100 回の試行を行い、シミュレーション上でのノード数の変化を見る。
なお、今回はモデルサイズの 4 分の 1 の、250 だけずらした。

表 1　シミュレーションパラメータ

モデルサイズ	1000×1000
ノード数	500、50000
通信半径	100、250、500
試行回数	100

4.3　実験結果

実験結果を示す。

図 8 はノード数 500、通信半径 250 の場合の結果である。

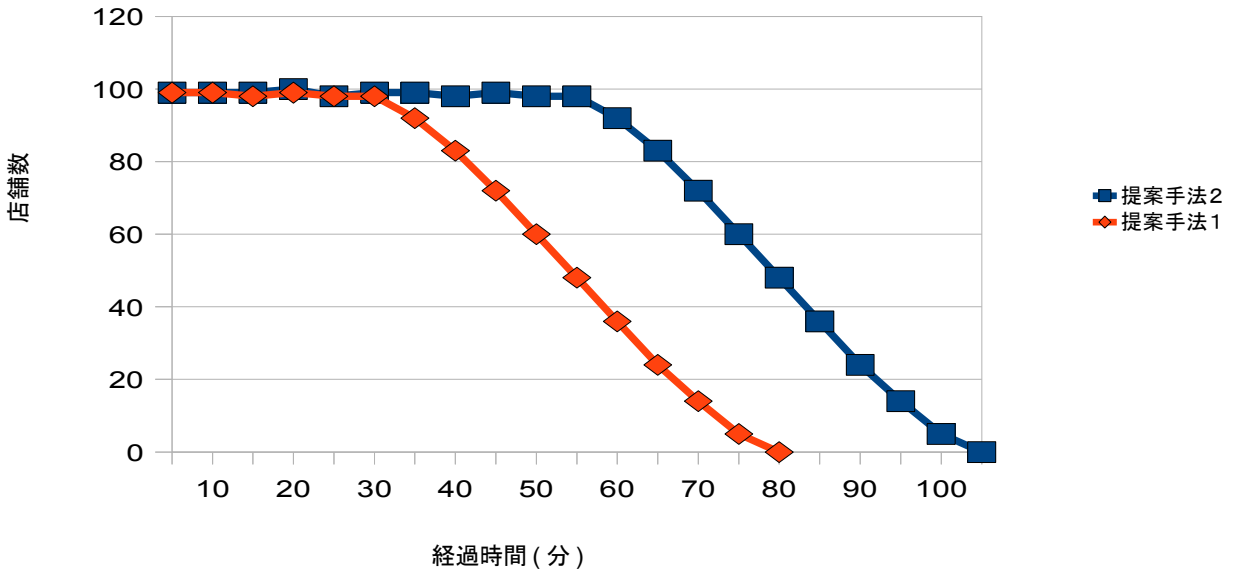

図 8

図 9 はノード数 50000、通信半径 250 の場合の結果である。

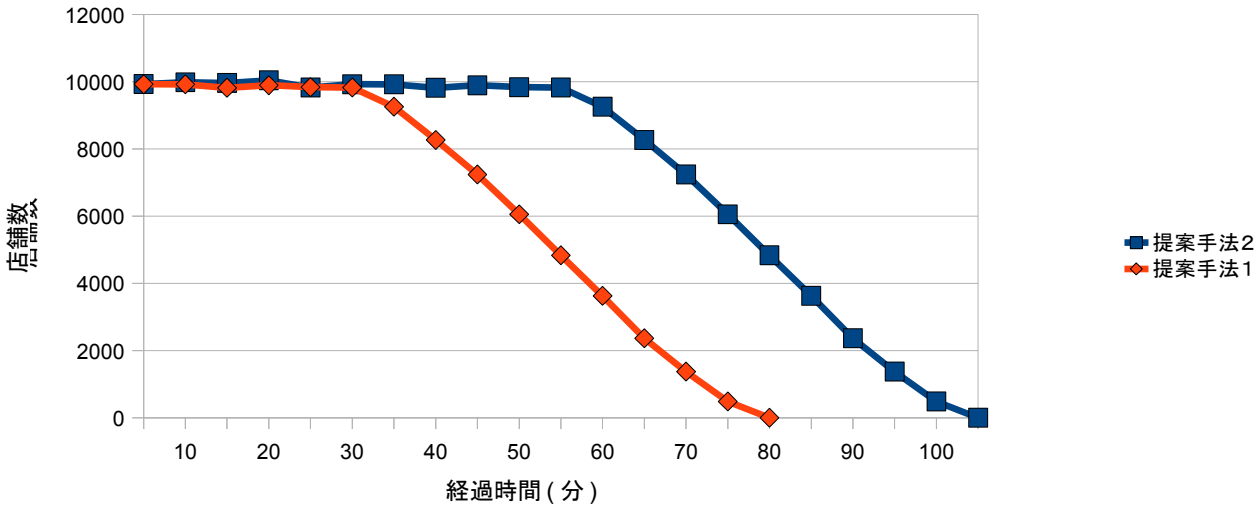

図 9

図 10 はノード数 50000、通信半径 100 の場合の結果である。

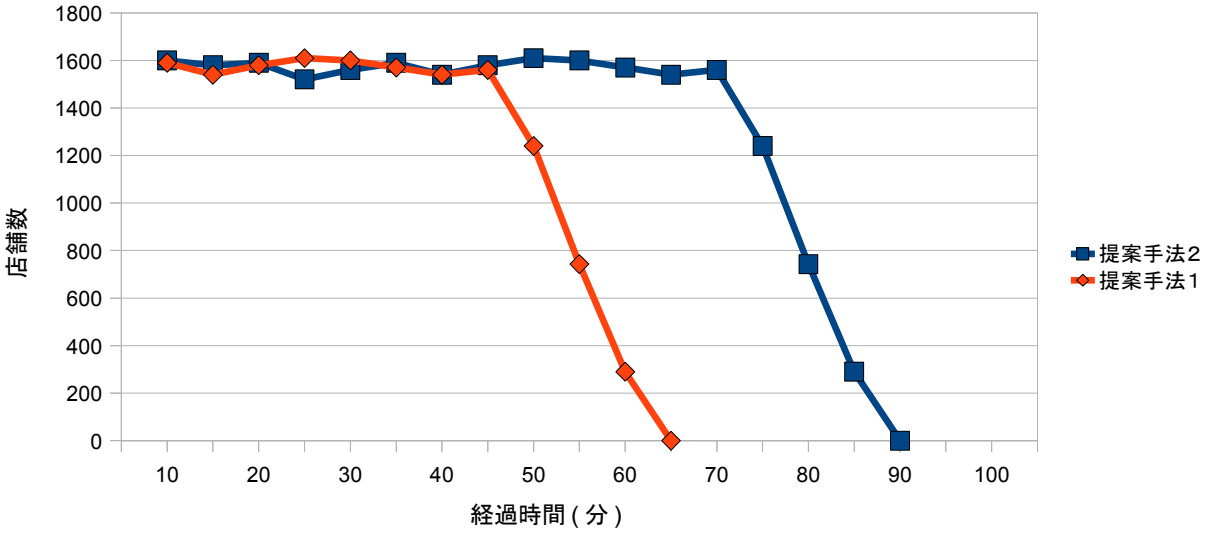

図 10

図 11 はノード数 50000、通信半径 500 の場合の結果である。

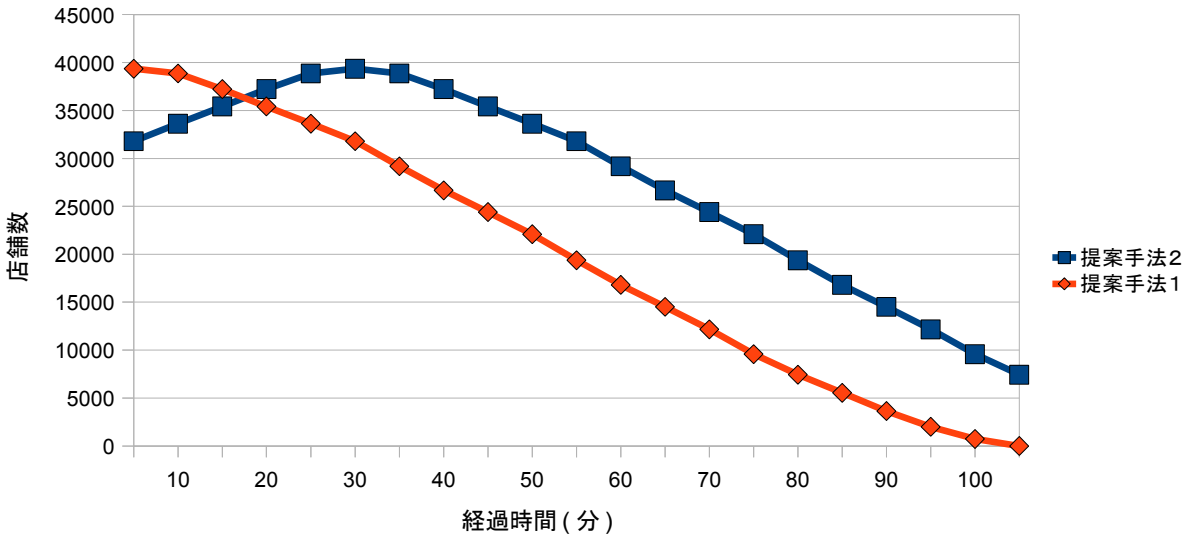

図 11

第五章　結果・考察

本論文では、自分の位置情報から周りの店の情報を入手することが出来、歩いている時に自動的に方向を推定し、その方向の情報を持っている人から店舗情報を手に入れるシステムを提案した。
実験結果から、位置を推定することで、そうでないものより店舗情報数が減少するまでの時間を長くする事が出来た。
今後はシミュレーションだけでなく、実際に地図やGPSを用い、実験・検証を行う。

謝辞

本研究を進めるにあたり、ご指導いただいた龍谷大学情報メディア学科の三好力教授に深く感謝いたします。また、日常の議論を通じて多くの知識や示唆を頂いた同研究室の皆様や多大なご協力を頂いた皆様に深く感謝いたします。

最後に、支えてくれた友人に深く感謝いたします。

参考文献

NTT　未来ねっと研究所
http://www.ntt.co.jp/mirai/organization/organization0305.html

ドコモ　i コンシェル提供サイト
http://iconcier.nttdocomo.co.jp/biz/

付録

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define SQUx 1000 //施設の範囲 x
#define SQUy 1000  //施設の範囲 y
#define NODE 500//ノードの総数
int main(void)
{
	FILE *file2;
	int r,x,y,x1,y1,i,p;//ファイルから読み込んだ値の格納先
	float kr;
	//ランダムでのノード発生
	int kakunoux[NODE],kakunouy[NODE];
	for(i=0;i<NODE;i++){
		kakunoux[i] = 0;
		kakunouy[i] = 0;
	}



	//ランダムの初期化
	srand((unsigned) time(NULL));

	p=0;
	while(p!=NODE){
		kakunoux[p]=rand()%SQUx;
		kakunouy[p]=rand()%SQUy;
		p++;
	}

	////////////////////////////
	//printf("r=");
	//scanf("%d",&r);
	r = 250;
	file2 = fopen("kekka.txt","w+");
	printf("現在地の(x,y)座標を入力 :");
	scanf("%d,%d", &x1, &y1);
	for(i=1;i<NODE;i++){
		kr =sqrt(   ((kakunoux[i]-x1)*(kakunoux[i]-x1)) + ((kakunouy[i]-
y1)*(kakunouy[i]-y1))     ); //直線距離の二乗
		if(r>=kr){
			fprintf(file2,"(%d,%d)",kakunoux[i],kakunouy[i]);
			fprintf(file2,"距離＝%lf\n",kr);
			printf("(%d,%d)",kakunoux[i],kakunouy[i]);
			printf("距離＝%lf\n",kr);
		}
	}
	fclose(file2);
	return 0;
}



//歩行シミュレーター


import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Image;



//-------------------------------------------------------

public class Myapplet extends Applet implements Runnable {

int NX; // X 方向の迷路ブロック個数。奇数になるようにする。
int NY; // Y 方向の迷路ブロック個数。奇数になるようにする。
int NX2; // = ((NX - 1) / 2) - 1;
int NY2; // = ((NY - 1) / 2) - 1;
int goal_x; // = NX - 2;
int goal_y; // = NY - 2;

final int SIZE = 10; //ブロックサイズ(ピクセル数)
final int HALF = SIZE / 2;
final int start_x = 1;
final int start_y = 1;
final int space = -1;
final int wall = -2;
final int visited = 4; // これに更に移動の向き(0～3:下記の意味)を加算。
final int d_right = 0;
final int d_down = 1;
final int d_left = 2;
final int d_up = 3;
int dx[] = {0, 0, 0, 0};
int dy[] = {0, 0, 0, 0};
Color start_color;

Thread thread = null;
Dimension size;

Image back;
Graphics buffer;
final Color dark_green = new Color(0x20, 0x80, 0x20);

int block[][];

int stage;
int now_x; // 現在位置
int now_y;
int now_d; // 移動の向き: 0～3
int old_x; // 1 つ前の位置
int old_y;

int delay; // ミリ秒単位の歩行中 delay。

//-------------------------------------------------------

public void init() { // 初期化処理。

String s = getParameter("delay");
if (s != null) {
delay = Integer.decode(s);//.IntValue();
} else {
delay = 1;
}

size = getSize();

// 実行画面の実際のサイズから以下の値を決める。

NX = size.width / SIZE;
NY = size.height / SIZE;
if ((NX % 2) == 0) NX--; // 奇数にする。
if ((NY % 2) == 0) NY--; // 奇数にする。

size.width = NX * SIZE;// 変えると黒枠延長
size.height = NY * SIZE;

NX2 = ((NX - 1) / 2) - 1;
NY2 = ((NY - 1) / 2) - 1;
goal_x = NX - 2;
goal_y = NY - 2;

back = createImage(size.width, size.height);
buffer = back.getGraphics();

block = new int[NX][NY];
makeMaze();

// スレッド始動。

stage = 0;

thread = new Thread(this);
thread.start();
}
//-------------------------------------------------------

private void makeMaze() {

// 全体をクリア。

for (int x = 0; x < NX; x++) {
for (int y = 0; y < NY; y++) {
block[x][y] = space;
}
}

// 外枠をセット

for (int x = 0; x < NX; x++) {
block[x][0] = wall;
block[x][NY-1] = wall;
}
for (int y = 0; y < NY; y++) {
block[0][y] = wall;
block[NX-1][y] = wall;
}

// 基準点をセット。

for (int x = 1; x <= NX2; x++) {
for (int y = 1; y <= NY2; y++) {
block[x * 2][y * 2] = wall;
}
}

// 迷路作成。

decide_right_or_left();

for (int x = 1; x <= NX2; x++) {
for (int y = 1; y <= NY2; y++) {
if (y == 1) {
int d = (int)(Math.random() * 4);
block[x * 2 + dx[d]][y * 2 + dy[d]] = wall;
```

```
} else{
boolean flag = true;
while (flag) {
int d = (int)(Math.random() * 3);
if (block[x * 2 + dx[d]][y * 2 + dy[d]] == space) {
block[x * 2 + dx[d]][y * 2 + dy[d]] = wall;
flag = false;
}
}
}
}
}
now_x = start_x;
now_y = start_y;
now_d = 0;
old_x = start_x;
old_y = start_y;
block[start_x][start_y] = visited;

// 乱数で、壁を触るのが右手か左手かを決める。

decide_right_or_left();
}
//---------------------------------------------------------

public void decide_right_or_left() {

if ((((int)(Math.random()) * 100) % 2) == 0) {
// 右手。
dx[0] = 1; dy[0] = 0;
dx[1] = 0; dy[1] = 1;
dx[2] = -1; dy[2] = 0;
dx[3] = 0; dy[3] = -1;
start_color = Color.red;
} else {
// 左手。
dx[0] = -1; dy[0] = 0;
dx[1] = 0; dy[1] = -1;
dx[2] = 1; dy[2] = 0;
dx[3] = 0; dy[3] = 1;
start_color = Color.yellow;
}
}
//---------------------------------------------------------

public void update(Graphics g) {
paint(g);
}
//---------------------------------------------------------

public void paint(Graphics g) {

if (stage == 2) {
draw_start_and_goal();
g.drawImage(back, 0, 0, this);
return;
}

// 描画領域をまず黒にする。

buffer.setColor(Color.black);
buffer.fillRect(0, 0, size.width, size.height);

// 壁を白で表示。

for (int x = 0; x < NX; x++) {
for (int y = 0; y < NY; y++) {
if (block[x][y] == wall) {
buffer.setColor(Color.white);
buffer.fillRect(x*SIZE, y*SIZE, SIZE, SIZE);
}
}
}
// 外枠を描く。

buffer.setColor(Color.blue);
buffer.drawRect(0, 0, NX * SIZE - 1, NY * SIZE - 1);

if (stage == 1) { // 迷路内を歩行中:

// 探索範囲とベストな足跡(これまでの最短経路)を表示。

for (int x = 1; x < NX-1; x++) {
for (int y = 1; y < NY-1; y++) {
if (block[x][y] < 0) continue;
draw_rect_at_point(x, y, dark_green, null);
if (block[x][y] < visited) continue;
show_direction(x, y, 0, Color.green, 0);
}
}

// 直前の位置を深緑で、現在の位置を緑で表示。

draw_rect_at_point(old_x, old_y, dark_green, null);
draw_rect_at_point(now_x, now_y, Color.green, Color.black);

// 直前の進行方向を黒で、現在の進行方向を赤で表示。

int old_d = block[old_x][old_y] - visited;
if (old_d >= 0) {
show_direction(old_x, old_y, old_d, Color.green, 0);
}
show_direction(now_x, now_y, now_d, Color.red, 4);
}
draw_start_and_goal();
g.drawImage(back, 0, 0, this);
}

//---------------------------------------------------------

private void draw_start_and_goal() {

draw_rect_at_point(start_x, start_y, start_color, Color.white);
draw_rect_at_point(goal_x, goal_y, Color.blue, Color.white);
}

//---------------------------------------------------------

private void draw_rect_at_point(int bx, int by, Color fill, Color line) {

// ブロック升目(bx, by)を fill カラーで塗り、内側に line カラーで矩形を追加。

int x = bx * SIZE;
int y = by * SIZE;

buffer.setColor(fill);
buffer.fillRect(x, y, SIZE, SIZE);

if (line != null) {
buffer.setColor(line);
buffer.drawRect(x + 1, y + 1, SIZE - 3, SIZE - 3);
}
}

//---------------------------------------------------------

private void show_direction(int bx, int by, int direction, Color fill,
int shift) {

// 進行方向を小さい矩形で表示: (x, y)は矩形の左上コーナー座標。

int x = (bx*SIZE + HALF) + (dx[direction] * shift) - 3;
int y = (by*SIZE + HALF) + (dy[direction] * shift) - 3;

buffer.setColor(fill);
buffer.fillRect(x, y, 7, 7);
}
//---------------------------------------------------------

public void run() {

while (true) {
switch (stage) {
case 0:
makeMaze(); // 迷路作成。
repaint();

sleeping(200); // 2000 ミリ秒待機する。

stage = 1;
break;

case 1: // 迷路の中を歩いている途中。

sleeping(delay); // 指定時間のあいだ待機する。

move();
repaint();

if ((now_x == goal_x) && (now_y == goal_y)) {
// ゴールに着いたら最終経路を表示。
stage = 2;
draw_final_route();

// 5000 ミリ秒待機する。
sleeping(5000);

// また次の迷路処理へ続く...

stage = 3; //ループ停止
          //0; //ループする
}
break;

default:
break;
}
}
}
//---------------------------------------------------------

private void sleeping(int milliseconds) {

try {
Thread.sleep(milliseconds);
} catch (InterruptedException e) {
}
}
//---------------------------------------------------------

private void move() {
```

```
// 一歩ずつ、次を確かめながら進む。

int old_d = now_d;

// 右(または左)、進行方向、左(または右)、後方(進行方向の逆)の
// 順でチェックする。
// これでダメならば、全くどちらの方向にも進めないが、
// そういうことは無いであろう。

for (int delta_d = 1; delta_d >= -2; delta_d--) {
int next_d = (now_d + delta_d + 4) % 4;
int next = block[now_x + dx[next_d]][now_y + dy[next_d]];

if (next != wall) {
now_d = next_d;

old_x = now_x;
old_y = now_y;
now_x += dx[now_d];
now_y += dy[now_d];

if (block[old_x][old_y] >= visited) {
old_d = block[old_x][old_y] % visited;
} else {
old_d = space;
}
if (clear_visited_flag(old_d, now_d, old_x, old_y) == false) {
block[now_x][now_y] = visited + now_d;
}
return;
}
}
}
//---------------------------------------------------------

private boolean clear_visited_flag(int old_d, int now_d, int x, int y) {

// (x, y)で指定された場所で後戻りが生じていれば、visited フラッグを消す。

if (old_d == space) return false;
if (block[x][y] < visited) return false;

int value = 0;
if (block[x-1][y] >= visited) value += block[x-1][y];
if (block[x+1][y] >= visited) value += block[x+1][y];
if (block[x][y-1] >= visited) value += block[x][y-1];
if (block[x][y+1] >= visited) value += block[x][y+1];

if (value >= visited * 2) return false;

if ((old_d != now_d) && ((old_d + now_d) % 2 == 0)) {
block[x][y] %= visited;
return true;
}
return false;
}
//---------------------------------------------------------

private void draw_final_route() {

// ゴール直前点を最終経路描画に必ず含める。

buffer.setColor(Color.green);
buffer.fillRect(old_x*SIZE, old_y*SIZE, SIZE, SIZE);
block[old_x][old_y] += visited;

block[start_x][start_y] = visited;
block[goal_x] [goal_y] = visited;

// 最短経路を青ベタと黒線で表示。

buffer.setColor(Color.black);
for (int x = 1; x < NX-1; x++) {
for (int y = 1; y < NY-1; y++) {


if (block[x][y] < visited) System.out.println("["+x+","+y+"]");//確認用
if (block[x][y] < visited) continue;
// 緑ベタ:

buffer.setColor(Color.blue); //最短経路の表示色
buffer.fillRect(x*SIZE, y*SIZE, SIZE, SIZE);

// 黒線: 中心から HALF サイズで２本ずつ

int center_x = x*SIZE + HALF;
int center_y = y*SIZE + HALF;
buffer.setColor(Color.black);

if (block[x-1][y] >= visited) {
buffer.drawLine(x*SIZE, center_y, center_x, center_y);
}
if (block[x+1][y] >= visited) {
buffer.drawLine(center_x, center_y, (x+1)*SIZE-1, center_y);
}
if (block[x][y-1] >= visited) {
buffer.drawLine(center_x, center_y, center_x, y*SIZE);
}
if (block[x][y+1] >= visited) {
buffer.drawLine(center_x, center_y, center_x, (y+1)*SIZE-1);
}
}
}
repaint();
}
//---------------------------------------------------------

}
```