Stochastic Diffusion Search
A Basis for A Model of Visual Attention?

by

Robert Summers

(Student No.: 0297 0472 9)

Department of Communication and Neuroscience

Keele University

Keele

Staffordshire

ST5 5BG

September 1998

Acknowledgements

I would like to thank Dr. Stephen Westland for suggesting the project in the first place, guiding me through it and reading my rough draft and also Mark Bishop and Slawomir Nasuto at Reading University Cybernetics Department for their invaluable assistance at the beginning.

This research was funded by a grant from the EPSRC.

"**Febrius.** Ephebian philosopher. He proved that light travels at about the speed of sound in his famous 'Give us a shout when you see it, OK?' experiment involving two hills, a lantern with a movable cover over it and an assistant with a very loud voice."

The Discworld Companion - Terry Pratchett and Stephen Briggs.

Table of Contents

Abstract

The various properties of Stochastic Diffusion Search (SDS) have led to it being mooted as a possible basis for a model of visual attention. Its parallel architecture links in well with the architectures proposed in theories of human visual search, notably *guided search* (Wolfe and Cave, 1989). Factors affecting the performance of SDS such as model size, search space size, number of agents, search space similarity, noise and termination conditions were all investigated. The mean convergence time of SDS was found to be more heavily dependent on the number of agents, search space similarity, noise and termination conditions, rather than on the size of the search space and the model. The psychophysical plausibility of 1D SDS was tested and results were obtained which were analogous to oriented-line-target detection and mimicked human data. A 2D implementation of the algorithm was produced and tested on a small number of experiments some of which have been previously tested on humans such as line target detection. Results did not mimic human data. However, the response of the algorithm was low when testing the response of a vertical filter 2D SDS to a display of vertical elements. This was expected and could be explained by the distribution of activity across a wide area of the search space which would result in a lower overall response. The conclusions that can be drawn from this is that much further research should be directed towards SDS as it has many interesting properties that tie in with visual search.

Introduction

An understanding of the computational processes of the early visual system would be of great help to many computer scientists in the field of vision. Computers are notoriously bad at finding objects in a scene if they are partially obscured or deformed in some way. Humans, on the other hand, find the task relatively simple. A computational model of the human visual system might, therefore, give rise to better computational vision systems.

Although there are many algorithms associated with object location and recognition, such as the Hough transform, they are invariably computationally expensive.

Stochastic Diffusion Search is a parallel-based algorithm that has some interesting properties which could make it suitable for use as the basis for a computational model of visual search.

Chapter 1 outlines the experimental results and theories underlying human visual search before introducing the stochastic diffusion search algorithm. Chapter 2 describes the factors that may affect the performance of stochastic diffusion search which are investigated in detail in chapter 3. The application of stochastic diffusion search to human visual search tasks is outlined in chapter 4. Finally the conclusions, limitations and areas for further research are discussed in chapter 5.

Chapter 1

Literature Review

1-1 Visual Search

Introduction

Understanding the visual world is an enormously complex task from a computational viewpoint. Humans, however, appear to find the task effortless. If placed in a room, previously unknown, a normal adult human would have little difficulty in organising and recognising the objects contained within into a logical structure. This would also be done almost instantaneously. The methods required to detect and recognise objects such as edge extraction, colour perception, movement and distances pass by unnoticed.

Firstly, I will explain the experimental paradigm. Secondly I will outline the results and their interpretation which have led to the means of defining features and theories attempting to explain visual search.

The Experimental Paradigm

In the experiments conducted by Treisman and Gelade (1980), Wolfe et al. (1989), Wolfe (1994, 1996) and Foster and Westland (1995) subjects would be asked to determine the presence or absence of a target item amongst a number of distractor items on a visual display unit (see Figure 1-1). Therefore two conditions exist, target present and target absent. For example the target could be a vertical line amongst a number of tilted lines (the distractors) of a fixed or differing orientation.

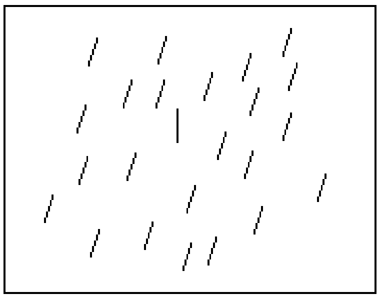

Figure 1-1: A typical display for target present condition.

Typical performance measures include reaction times (RT) and accuracy of the responses; e.g. percent correct or d'. The measure d' combines hit rates and false alarms to generate bias-free performance classifications. Results are often analysed in terms of RT x set size slopes (where set size is the total number of items in the display) and the type of search strategy employed are often deduced from this (Wolfe, 1996). Two sets of data stem from the trials, one for target present searches, and one for target absent searches.

If the level of accuracy as a function of presentation time is to be determined then the visual display is only presented for a short time. The search is terminated by a random display (or mask) which is assumed to displace any after-image in the retina, (Wolfe, 1996). Thus, the visual search cannot continue once this mask has been displayed. The display time, known as the *stimulus onset asynchrony* (SOA) can be altered. An Accuracy x SOA slope is then produced for both conditions. This type of experiment can also be used to explore the mechanisms of the early visual system in terms of its ability to detect and/or localise a target with short presentation times.

Interpretation of Results

Treisman and Gelade (1980) and Treisman (1986) proposed that a search-type dichotomy existed depending on the properties of the target and the distractors in the search task. Based on the *primal sketch* theory of Marr (1982) they also hypothesised about the coding of images in the early visual system.

If the target and distractors were defined by a single feature (e.g. a vertical line amongst tilted ones) and the difference between the orientation of the distractors and the target was above a discrimination threshold then the search time would not be affected by number of distractors, it would simply *pop out*. Treisman and Gelade (1986) inferred that the search mechanism was parallel as all the items could be sufficiently processed together to determine the presence of a target.

However, a search for a 2 among 5 's (Wolfe, 1996) was inferred to involve a serial mechanism. The reason being that in target present trials the RT x set size slopes were approximately 20-30 ms/item. That is that for every distractor (5) added to the display an extra 20-30 ms was required, on average, to find the target. In target present conditions a serial search would, on average, have to traverse half the items in the display before finding the target. That is, it could be the very first item to be looked at, or it could be the very last item. Therefore the RT x set size slopes found by Treisman and Gelade (1980) of 20-30 ms/item reflect a 40-60 ms/item serial search. The hypothesis from this is that target absent trials will result in RT x set size slopes of 40-60ms/item as every item will have to be examined before the search is terminated. Treisman and Gelade (1980) concluded from this that a 2:1 RT x set size slope ratio between target absent and target present trials indicated that serial mechanisms were involved. They found that simple feature searches, e.g. a vertical line amongst oriented lines, required parallel searches whilst searches involving items with conjunction of two features, e.g. a red vertical line amongst blue vertical and red horizontal lines, required serial search. The 2 among 5 's search would be another such conjunction search. The target and distractor items share a number of properties, three horizontal lines and two vertical lines. It is only the relative positions of the vertical lines that are transposed; this is why the target is hard to detect.

However, as Wolfe (1996) states, “Inferring mechanisms from slopes is not that simple”. There are a number of factors that have led most researchers in the field to discard the serial/parallel dichotomy of Treisman’s Feature Integration Theory, not least Treisman herself (Treisman, 1993). However, although the idea of a serial/parallel dichotomy has been (mostly) discarded it does not mean that visual search is either entirely serial or entirely parallel. Evidence suggests that there is a parallel search strategy working with a serial strategy (Wolfe et al., 1989; Wolfe, 1994 and 1997).

Wolfe (1996) notes that only the original experiments conducted by Treisman and Gelade (1980) appear to show a serial/parallel dichotomy; “The evidence [from further studies] shows a continuum of search results”. Later studies by Treisman (Treisman and Sato, 1990) showed much shallower RT x set size slopes for conjunction searches. The reasons for obtaining such vastly different results are hard to fathom. More recent experiments have used a visual display unit to carry out the experiments (e.g. Wolfe et al., 1989) whilst Treisman’s original experiments used hand drawn slides presented via a tachistoscope. Treisman’s slides had a white background whereas the display of Wolfe’s (and others) had a black background. Wolfe et al. (1989) noted that their shallower slopes could have been due to the differing salience of the target and distractor items between their experiment and that of Treisman and Gelade (1980). Wolfe et al. (1989) tested the theory and emulated the original experiments. Indeed, the results indicated steeper RT x set size slopes, in some cases enough to account for the difference between the results obtained by Treisman and Gelade (1980) and later experiments. However, this does nothing to enhance the cause for a serial/parallel dichotomy rather, it suggests that target/distractor salience (or contrast) is another basic feature of the early visual system.

If certain results are due to self-terminating serial searches then why are errors made? That is, why is a target sometimes missed when it exists and why are they sometimes “found” when they do not? The 2:1 slope ratio hypothesis assumes that the serial self-terminating search is “exhaustive” in that for target absent trials every item is checked. So no provision is made for false alarm errors in target absent trials. Furthermore, Zenger and Fahle (1997) noted that missed targets occur more often than false alarms. Both of these points “complicate” the 2:1 slope ratio hypothesis (Chun and Wolfe, 1996). This is because a number of false alarms will bring the average search time down for target absent trials whilst missed targets will increase the average search time in target present trials.

Essentially, continuing research with a serial/parallel dichotomy in mind is not supported by the evidence. Wolfe (1996) argues instead that the continuum of search slopes found can be “described neutrally in terms of search efficiency” (see figure 1-2).

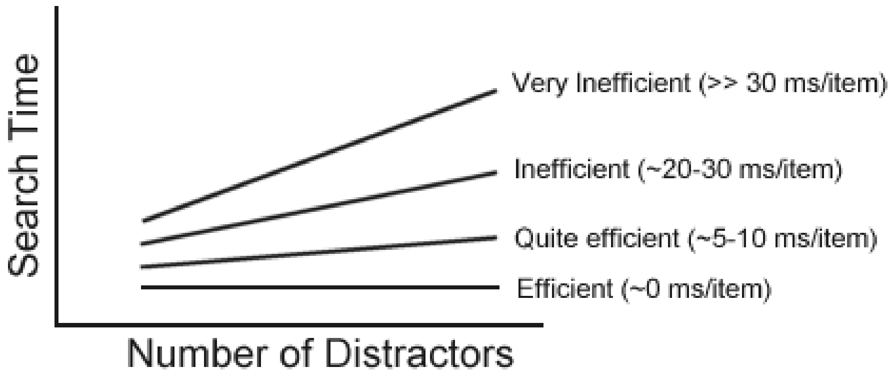

Figure 1-2: The continuum of search slopes defined in terms of efficiency, after Wolfe (1996).

The many accepted theories of visual search, and guided search in particular, depend on parallel-guided serial search process. These theories will be discussed after the notion of features in visual search has been examined.

What is a feature?

The choice of target and distractor items is not a random process. Much of the choice is based on physiological evidence. For example the existence of orientation-selective cells in the visual cortex (Hubel and Wiesel, 1968) gave rise to experimental displays of the type shown in figure 1-1. The object is to determine the minimum difference in orientation that the eye can discern “efficiently”. The definition of efficiency in this context will be discussed later on.

In the early days of visual search experiments a feature was defined if it had the property that its RT x set size slopes had almost zero gradient (Wolfe, 1996), i.e. it was found using the so-called parallel search mechanism. That is a target would “pop-out” from the search space in a given a time, regardless of the number of distractors (Treisman and Gelade, 1980). However, since the idea of a parallel/serial dichotomy has been almost laid to rest this is a far from adequate definition.

Treisman (1986) argued also that real world objects comprise a number of properties, particularly of texture, whose boundaries “show a continuity of lines or curves” which are in essence part of the same cluster. Therefore texture segmentation is an important part of feature integration.

Basic Features in Visual Search

There are number of generally accepted basic features in visual search as well as a number of controversial ones. A non-exhaustive list would include:

- orientation;
- curvature;
- colour;
- motion;
- depth;
- stimulus salience.

However, this project is only concerned with orientation as a feature and therefore the other features are out of the scope of this research.

Although the notion of orientation as a basic feature is backed up by physiological evidence (Hubel and Wiesel, 1968), no such cells exist for curvature but it is widely acknowledged as a basic feature. Treisman (1986) gives a clue as to why this is the case. She concludes that it is the presence of orientation and the presence of curvature that is coded. Simply, a vertical straight line is coded as a null value and it is only deviations from straightness and verticalness that are coded. This hypothesis arises from results of experiments with displays such as those in figure 1-3.

In the left hand box of figure 1-3, the target, a circle with an intersected line, pops out no matter how many distractors are added. However, in the right hand box the search time for the target increases with the number of distractors in the display. Similar results were found for oriented lines. If an oriented line was the target, amongst a number of vertical lines, the search time was almost entirely independent of the number of distractors, whereas the search time for a vertical line target amongst oriented lines increased as the number of distractors increased.

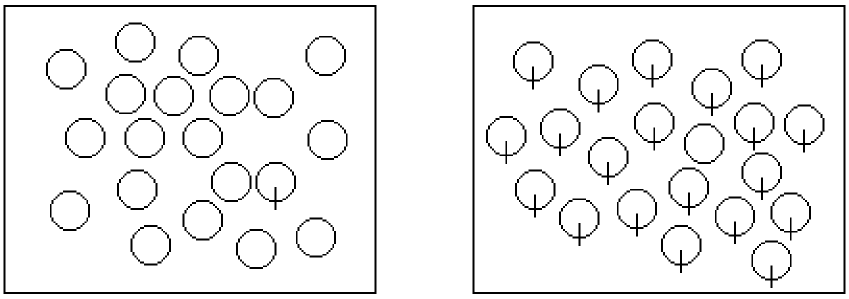

Figure 1-3: Displays used to show how the presence or absence of a feature, a vertical line in this case, affects the search time. (After Treisman, 1986)

Furthermore, Foster and Westland (1995) found evidence to suggest that at the pre-attentive stage there are two broadband orientation filters with peak sensitivities at the vertical and horizontal. These filters work before localisation of the target. If it is assumed that the early visual system employs a limited set of resources spread across the whole of the visual field then an interesting prediction comes to light; the response of the vertical filter would be high whilst the horizontal filter would be low. This is because the activity of the horizontal filter would be spread over the whole visual field whilst the vertical filter activity would be spread over a specific part of the visual field and therefore its activity would be concentrated in one area.

An outline of the Guided Search theory of Visual Search

The early Feature Integration Theory of Treisman and Gelade (1980) and the evidence refuting the idea of a serial/parrallel dichotomy led to Wolfe's model of Guided Search (Wolfe, Cave and Franzel, 1989; Wolfe, 1994; Wolf and Gancarz, 1997) now in its third incarnation and referred to as GS.

The GS model proposes that preattentive feature searches guide a serial search process, known as the *spotlight of attention*. Each feature is associated with an activity map. Each activity map is the sum of the response of the preattentive process for that particular feature at different areas in the visual field. The sum of the activity maps are taken and the spotlight of attention moves over the areas starting from those with the highest activity until the target is found.

Consider searching for a black vertical bar amongst black horizontal and white vertical bars. The activity maps from the preattentive processes for colour and orientation are summed, the one with the greatest activity in the summed map gives the location of the target (figure 1-4).

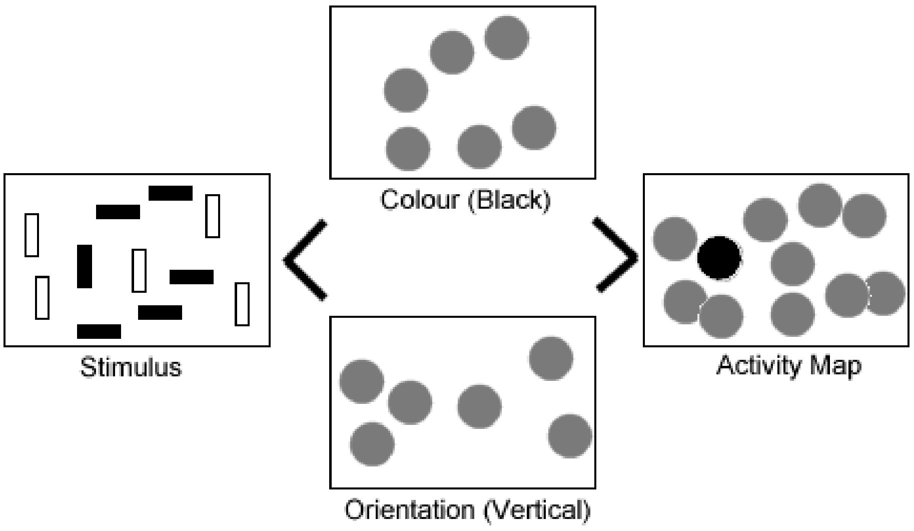

Figure 1-4: The overall activity map (right) that results from the summed activities of the preattentive processes for colour and orientation. The target in the stimulus is the black vertical bar and corresponds to the darkest patch in the activity map. This is the area of highest activity. After Wolfe (1997).

However, despite a widespread acceptance of GS there is still one major objection (Green, 1991). There is evidence to suggest that individual feature maps can not be suppressed whether or not they are currently active in the search task. Therefore for each visual search task the same number of feature maps are being summed. In theory the overall activity map should always take the same amount of time to compute. However, since different search tasks for individuals take different times to complete the notion of a pooled activity map based on the sum from all feature maps is a little harder to comprehend.

Green (1991) proposed that a network architecture, rather than the blackboard architecture of GS, satisfies the evidence better, (see figure 1-5). The architectures comprise features maps (FM) and activity maps. However in the network architecture the feature maps can communicate with each other and it is this communication that can sometimes hinder the search process.

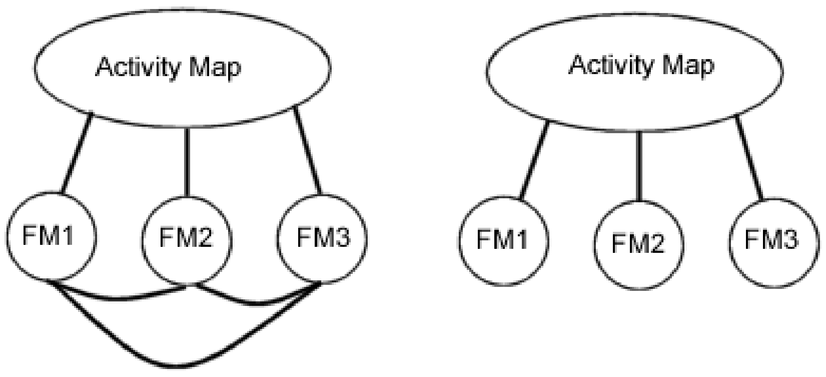

Figure 1-5: Two types of Feature Map architectures (Green, 1991), 'Network' on the left and 'Blackboard' on the right.

Despite this objection, the GS model is a good basis for a computational model of human visual search.

1-2 Stochastic Diffusion Search

Introduction

Terminology

SS	Search Space Size;	Q_N	Percentage of Noise in Model;
SS[x]	Value of xth element in the search space;	D	Number of Distractors in the model;
Q	Model Size;	D_S	Similarity of Distractors to the model;
Q[x]	Value of xth element in the model;	T	Threshold;
NA	Number of Agents;	BT	Base Threshold;
NA[x]	Mapping pointed to by the xth Agent;	ST	Stability Time;
A	Alphabet Size;		

Conventional neural networks have difficulty in classifying two or more distinct patterns to give the same output. These distinct patterns may actually be deformations - translations or rotations - of one specific pattern. This is known as the problem of stimulus equivalence or inexact matching.

A number of methods of solving this problem have been proposed, including Hinton Mapping and Fukushima's Neocognitron. These, unfortunately suffer from a high degree of computational requirements (Bishop 1988) and therefore are interesting rather than useful.

Stochastic Diffusion Search (SDS), developed by Bishop (1988), is a parallel based search algorithm that attempts to solve the problem of stimulus equivalence (or inexact matching) with a lower order computational method than that of the Neocognitron or Hinton mapping. It comprises a test phase and a diffusion phase.

Consider the situation of searching for a sequence of numbers (the model) inside a larger sequence of numbers (the search space). SDS randomly maps agents into the search space. An agent is an individual element (or unit) that holds a mapping, its current position in the search space. In the test phase each agent then compares its current mapping with a randomly selected mapping further along in the search space, but within the boundary of the length of the model. If this mapping is successful then the agent is labelled as being active. In the diffusion phase active agent's mappings are probabilistically diffused to other inactive agents. Agents which remain inactive are randomly assigned a new mapping. These processes are repeated until all, or most, of the agents have converged to the same mapping.

Following, is a more detailed explanation of SDS with a trivial example that is worked through in order to clarify the process.

The Stochastic Diffusion Algorithm.

Initialisation Phase.

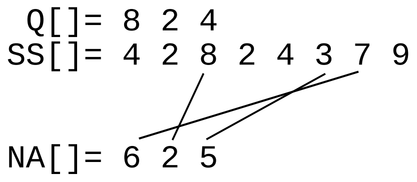

Figure 1-6: Initialisation Phase.

In the initialisation phase each agent is randomly assigned a mapping into the search space (figure 1-6). The elements of the model, search space and agents will be referred to as Q[0], SS[3], NA[2] etc. Whereas, Q[x] and SS[x] refer to an actual value of the xth unit in the model or search space respectively, NA[x] is the current position in the search space occupied by the xth agent. Note that NA[1]=2 is actually pointing to the first element in the model, in general this does not happen. After the initialisation phase occurs, a test phase and diffusion phase are repeatedly cycled through until the algorithm converges.

Test Phase.

```
          NA[] = 6 2 5
Random Element = 2 1 2
     Active Y/N? N Y N
```

Figure 1-7: Test Phase.

In the test phase each agent is assigned a random element, e, in the range from zero to the last position in the model, in this example between 0 and 2. The xth agent is considered to be active if and only if the value of the xth element of the model is equal to that of the search space 'e' positions along from where the xth agent is pointing. So in this case:

NA[0] is **inactive** because {Q[2] = 4} ≠ {SS[6+2] does not exist};

NA[1] is **active** because {Q[1] = 2} = {SS[2+1] = 2};

NA[2] is **inactive** because {Q[2] = 4} ≠ {SS[5+2] = 9}.

This comparison is a sampling of the search space that determines if the current mapping is a likely candidate for the correct position of the model. Each agent will only become or remain active if and only if the values of the search space at the current mapping and its offset match the equivalent values in the model.

Diffusion Phase.

```
           Agent: 0 1 2
       Try Agent: 2 * 1
Potentially Ok? N * Y
     New Mapping: 4 * 2
```

Figure 1-8: Diffusion Phase.

In the diffusion phase each inactive agent checks another randomly selected agent to determine if that agent is active or not. If it is then the inactive agent points to the mapping of the active agent and becomes active. If it is not then it randomly selects another mapping (figure 1-8). So in this case:

NA[0] checks NA[2] which is currently inactive therefore selects a mapping, 2;

NA[1] is active, therefore its mapping remains unaltered;

NA[2] checks NA[1] which is active and points to the mapping of NA[1];

Algorithm Termination.

In conditions of no-noise (a perfect match of the model exists in the search space) the algorithm would terminate as soon as all the agents were active. In general however, conditions of no-noise do not exist and the algorithm may never terminate. Therefore Bishop (1988) used a combination of a threshold and stability criterion to determine when the algorithm terminated. Three values are set:

- **Threshold (T) -** when the number of active agents equals T then the algorithm terminates;
- **Stable Time (ST)** and **Base Threshold (BT)** - when the number of active agents is greater than BT for ST iterations the algorithm terminates.

The algorithm cycles through the test and diffusion phases until the termination conditions are reached.

Once the algorithm has terminated the model's position in the search space is taken as being the modal value of each agent's mapping.

Computer Implementation.

The algorithm can be easily implemented on a serial computer. The structure of such a program is shown in figure 1-9 (the full code can be found in the appendices).

```
initialise agents;
while (termination = false) do
     {
     test mappings;
     diffuse correct mappings;
     }
print mapping;
```

Figure 1-9: Program Structure.

1-3 SDS and Visual Search

The links between SDS and human visual search may not, at first, be obvious. However, there are a number of features that both share.

Fundamentally, they both have an underlying parallel architecture. Very little is known about the parallel mechanisms in human visual search but the notion of an attentional spotlight in GS has a linkage with the agents in SDS.

The agents in SDS perform some kind of attentional mechanism which is not dissimilar to the attentional spotlight. Active agents, ones which may have found the target, have the ability to pass on their activity to other agents. The agents are guided to areas by feedback from the search space which is disseminated from the active agents to the inactive agents. It could be said that the *attention* of the agents is drawn to particular areas of the search space, areas which best fit the target at any given moment.

The random, but probabilistically guided, nature of SDS is also quite attractive in terms of a model of visual attention. The saccadic eye-movements made during visual search begin in a seemingly random manner before homing in on the target. Although the saccades are not entirely random, there is some bias towards the centre of the visual scene (Wolfe and Gancarz, 1997), the idea of agents randomly choosing positions and testing for plausibility makes sense.

Lastly, the architecture of visual search suggested by Green (1991) is very alike the internal architecture of SDS where agents communicate with each other to produce a map of activity at every iteration.

In short, although no claims are made that SDS is a model of human visual attention, its architecture has some interesting properties that make it a worthwhile subject of investigation as a basis for a model of attention.

Chapter 2

Factors affecting SDS

The size of the search space and the size of the model seem likely candidates for affecting the performance of SDS. In a sequential search the time taken to complete the search increases linearly with the size of the search space. The behaviour of SDS with respect to the size of the search space and the model will be studied in this project.

The number of agents used does not have to be equal to the number of elements in the model. The ratio between the number of agents and the size of the search space will have a bearing on the time taken for the algorithm to converge. Precisely how this affects speed of convergence will be investigated.

In the example (page 12), at each iteration (i.e. completion of one test and one diffusion phase) each agent only compares a mapping with, and only attempts to diffuse successful mappings to, one other agent. Again, this need not be the case, and more than one comparison may further prevent the algorithm from being stuck in a local minimum, i.e. erroneously setting the activation of a particular agent.

The alphabet (A) is the range of values that each element in the search space and the model can take. The greater A, the quicker the convergence of SDS. This is because the wider the range of values the search space can take, the less likely there are to be pairs of elements that match part of the model. Consequently the algorithm is less likely to get stuck in a local minimum.

Following on from this is the idea of distractors. Distractors are sequences in the search space which have a degree of similarity with the model. If, as has been mooted, SDS is a psychophysically plausible algorithm regarding attention then the following results should hold:

- at or below a certain degree of similarity, the number of distractors in the search space will not affect the time taken to complete the search;
- as the similarity of the distractors to the model increases search time will increase as more and more distractors are added to the search space.

Related to the notion of distractors is the problem of noise. A distractor can be thought of as noise in the search space; however, what happens when there is a model in the search space which is itself corrupted? There are two types of noise that can corrupt the model, *insertion noise* and *flipped-bits noise*. Noise that affects visual data could be flipped-bits noise where, for example, the target in the visual scene is not quite the same colour that was remembered. However, size and shape may not also be constant, and it is possible that insertion noise plays a part too. However since the implementation of SDS studied in this report works with a constant model size then the type of noise I will investigate will be flipped-bits noise. It would be expected that greater noise in the model will result in longer search time.

The termination conditions are quite arbitrary, and more than likely will need fine tuning depending on the amount of noise in the model. Investigation of these parameters will also be undertaken to determine whether or not a pattern can be established such that the algorithm can be self terminating.

The above experiments will not only consider Stochastic Search in its own right, but, where applicable will also compare the behaviour of SDS with that of an optimised sequential search.

Chapter 3

Analysis of SDS

3-1 Brief description of the Programs

An implementation of SDS was written in C++ (Appendix 1) as well as an optimised Sequential Search (Appendix 2). Each program allows a number of parameters to be varied such as search space size, model size, alphabet and number of trials per search space. The programs output a number of results files, one per search space size, which show the number of iterations to convergence, time to convergence and error, per individual trial. These files were then loaded into a spreadsheet where the results were analysed and graphs plotted.

The behaviour of the algorithm should be observed over a large range of search space sizes. Previous research (Bishop, 1988) had suggested that the algorithm worked best and had less performance-variance for larger model sizes. Therefore the range of search space sizes must be chosen in conjunction with a large model size. To obtain a reasonable idea of the algorithm's behaviour under different conditions each experiment needs to be run a sufficient number of times in order to determine average behaviour. Unless the behaviour of the algorithm due to a specific parameter was being investigated the following parameters were common for each experiment:

- Search space size: 1100 to 10000 in steps of 100;
- Model size: 1000;
- Number of trials per search space or model size: 1000;
- Alphabet: 10;
- Number of Agents: 1000;
- Termination Conditions: all agents active.

3-2 Experiment 1: The effect of Search Space Size

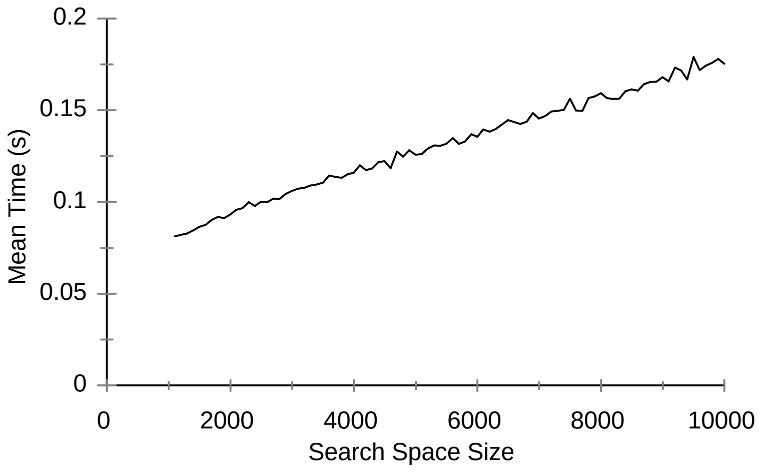

Figure 3-1: The effect of search space size on the convergence time of SDS.

Figure 3-1 shows a very clear linear relationship between the mean time to convergence and the size of the search space. The product-moment correlation coefficient r=0.9969 is, unsurprisingly, a statistically significant value. Note also that the slope of the line shows that when the search space doubles in size the corresponding convergence time does not. For example, at SS=5000 the mean convergence time is 0.1269spss (seconds per search space) whilst at SS=10000 the mean convergence time is 0.1753spss an increase by a factor of almost 1.4. The dependence on search space is not as heavy as might have been expected bearing in mind the same number of agents have twice the space to cover.

3-3 Experiment 2: The effect of Model Size

Figure 3-2 shows how the model size affects the performance of SDS. Strong positive linear correlations exist for each of four lines (r=0.99). As can be seen an increasing model size results in a greater mean convergence time. Figure 3-3 shows that for a given search space this relationship is linear. This is a surprising result since, for a given search space size, the larger the model size the fewer possible positions available in the search space. For serial searches this would mean less of the search space would need to be traversed whereas for SDS a greater proportion of the agents are likely to be active for a given iteration. However it should be noted that for this experiment the

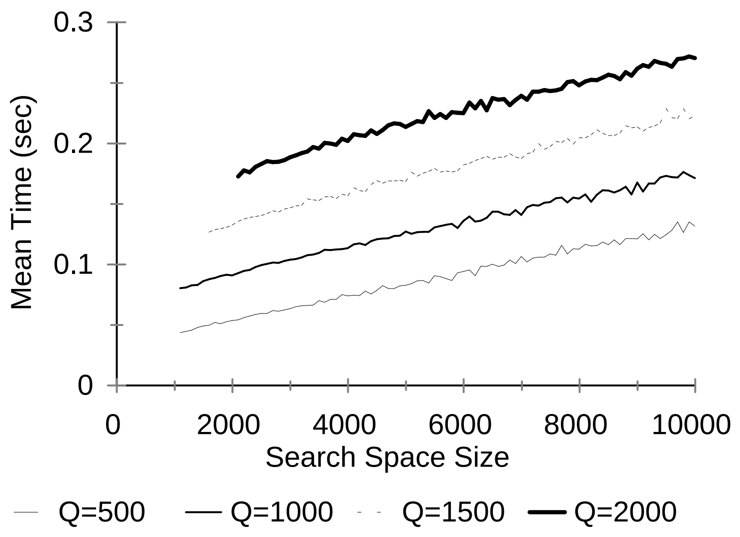

Figure 3-2: The effect of model size for different search space sizes on mean convergence time.

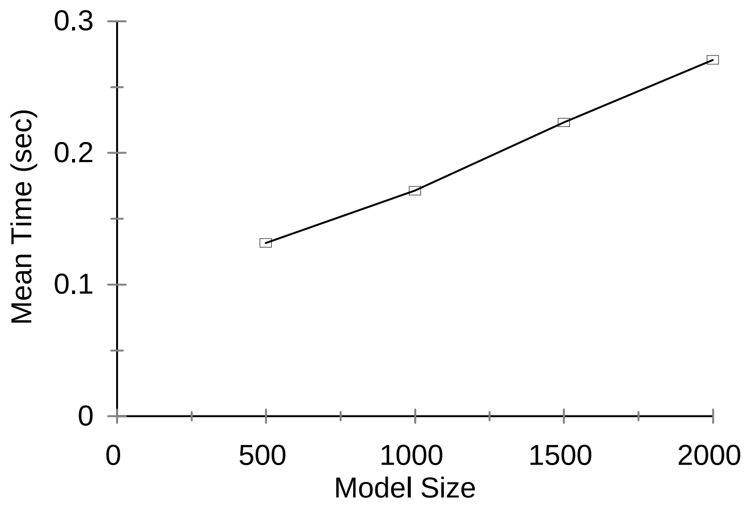

Figure 3-3: The effect of model size on mean convergence time for a constant search space size of 5000.

number of agents used was equal to the model size and therefore was not constant throughout. It is possible that the results obtained reflect more on the effect of the number of agents used rather than the model size.

Another experiment was conducted using a constant number of agents, 1000, a search space size of 10000 and model sizes of from 500 to 2000 in steps of 250. The results can be seen in figure 3-4. As expected the mean convergence time does decrease for larger search spaces. However, it exponentially decays from 0.28spss for a model size of 500 down to 0.18ssps for a search space size of 1250 where it appears to level off. It is likely that for search spaces much larger than 2000 the mean convergence time would decrease still further as during the initialisation phase many more agents would be active.

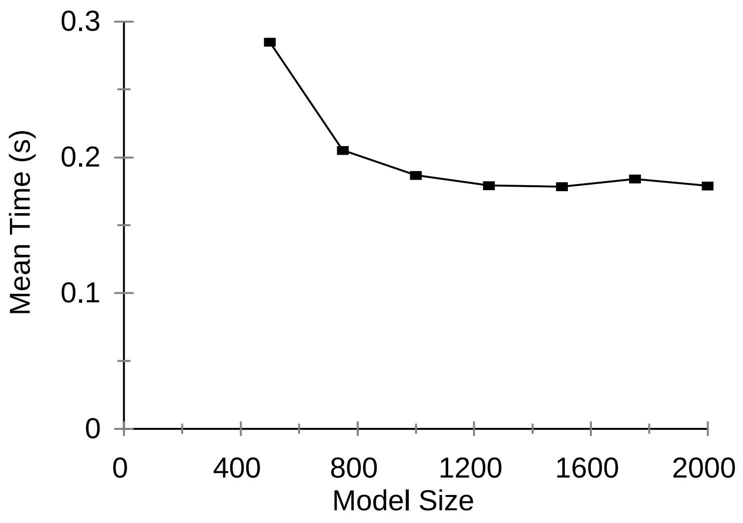

Figure 3-4: The effect of model size on mean convergence time for constant number of agents, 1000, and constant search space size, 10000.

3-4 Experiment 3: The effect of varying the number of Agents

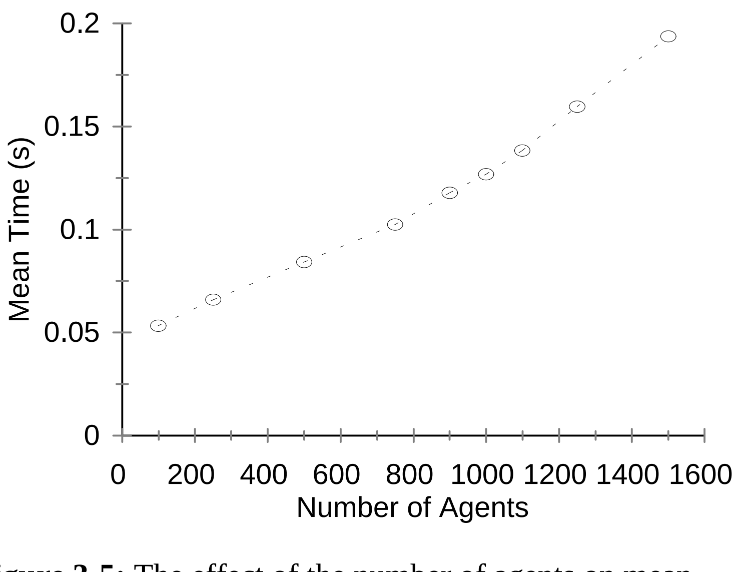

Figure 3-5: The effect of the number of agents on mean convergence time for a model of size 1000 and a search space of size 5000.

This experiment was run with a model size of 1000, a search space of size 5000 whilst varying the number of agents from 100 up to 1500. Figure 3-5 indicates that the mean convergence time increases with the number of agents. This agrees with the results of the initial experiment in the previous sub-section which suggested that it was the varying number of agents causing the rise in mean convergence time rather than the model size itself. This result appears initially surprising, for one would expect that fewer agents covering less of the search space would take longer to locate the model. However, whilst this is true in terms of time on a serial implementation of the algorithm a true parallel implementation would be quicker as more agents are used. In the serial implementation each agent is updated piecemeal whereas in a parallel implementation each agent would be computing its own activity at the same time. It is for this reason that the number of iterations, one cycle of the test and diffusion phases, is a more useful measure of the parallel performance of the algorithm. It must also be noted that in conditions of noise, when there is no perfect match of the model in the search space, fewer agents would be more likely to make an error in location.

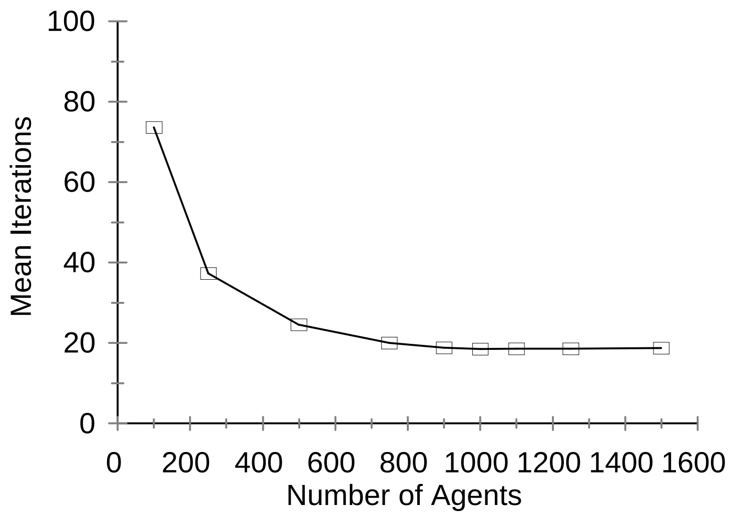

Figure 3-6: The effect of the number of agents on mean iterations to convergence.

Figure 3-6 shows how the number of iterations decreases as the number of agents increases. Performance does not appear to improve noticeably once the number of agents has exceeded the model length. This is contrary to what one might expect. However, although the more agents there are the greater the chance they will be active in any one iteration, there are also more agents who will be inactive and seeking out active agents with which to copy mappings. Therefore the hypothetical performance gain by using more agents is offset by the fact that a few extra iterations will be needed for all the agents to become active.

3-5 Experiment 4: The effect of varying the Alphabet.

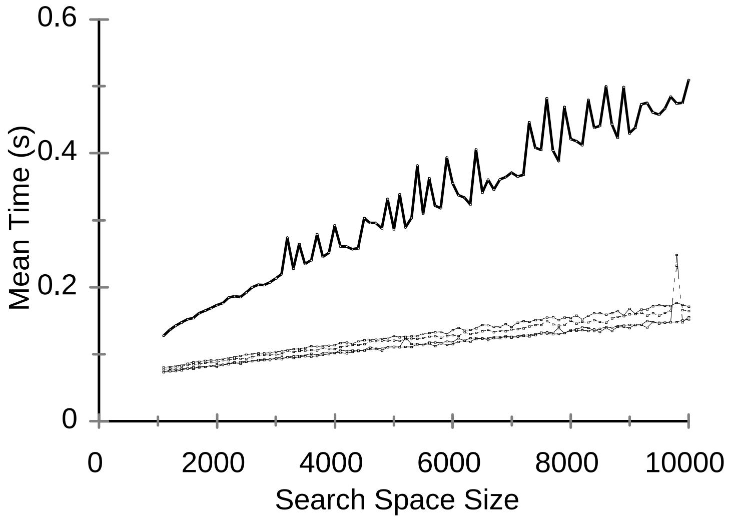

Figure 3-7: The effect of varying the alphabet on mean convergence time. The uppermost line corresponds to A=2. The other lines from top to bottom correspond to A values of 10, 16, 64 and 256 respectively.

As the ultimate aim of this project is to build a two-dimensional algorithm that performs some of the search tasks humans do it is important to test the performance of SDS with different alphabets. In the visual world the alphabet (A) corresponds to the number of gray scales of an image. It is for this reason that this experiment will test the performance of SDS with A values of 2, 16, 64 and 256 which correspond with both black and white images and typical grayscale levels found in computer images. These results are also compared with the value of A=10 which has been used in the other experiments. Figure 3-7 shows how when A=2 the mean convergence time is far greater and more variable than that for values of A greater than 10. This is because there are more partial matches of the model in a binary search space, especially when the SDS only samples the search space once per agent. Should the program be modified such that SDS samples a number of points at each test phase to determine the state of an agent’s activity then the algorithm might perform better. As many human visual search tasks require users looking at monochrome displays it is likely that serious thought will have to be given on how a two-dimensional SDS algorithm would search a binary search space for maximum performance and minimum variability. The other A values tested, 10, 16, 64 and 256, were not significantly different from each other. An F-test between the A values of 10 and 256 resulted in a probability of less than 5% that they were different.

One anomaly in the results, which can be seen in figure 3-7, occurred when SS=9800 and A=16 or 64. The search time almost doubled over the previous value before returning to a value more consistent with the linear results obtained thus far. No explanation for this springs to mind, especially when a rerun of the experiment for SS=9800 and A=16 failed to replicate the initial results. It seems that 1000 trials is not always enough to iron out statistical blips in the random number generator!

3-6 Experiment 5: The effect of Distractors

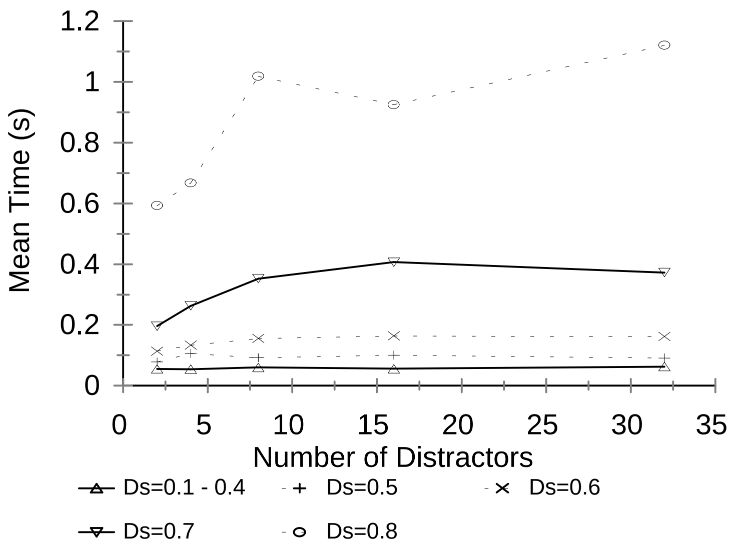

Figure 3-8: The effect of similarity and number of distractors on mean convergence time for a search space of size 5000 and a model size 100.

A new module was written in the SDS program (see Appendix 2) which creates search spaces in a carefully defined way. The search space is generated by adding a number of partial matches, distractors, of the model to the search space as well as one perfect match of the model. The distractor similarity, Ds, is measured in terms of the proportion of the model it matches. So for Ds=0.4 exactly 40% of the model is found in the distractor. The similarity of the distractors can be varied each time the program is run. Mean convergence times are computed for 2, 4, 8, 16 and 32 distractors. Figure 3-8 shows how the mean convergence time is affected by the number and similarity of distractors. For Ds values of 0.1 to 0.4 little difference can be found. However, as Ds increases the mean convergence time increases. Yet, for Ds less than 0.7 increasing the number of distractors does not increase the mean convergence time. For Ds's of 0.7 and 0.8 the mean convergence time increases as the number of distractors increases. This is evidence of the so-called psychophysical plausibility of SDS.

Consider the analogy of searching for a target oriented line amongst a number of other oriented lines of difference $\Delta\theta$ from the target. In this case Ds is equivalent to the value $\Delta\theta$, however larger Ds is equivalent to smaller $\Delta\theta$ and vice versa.. The shape of the curves show in figure 3-8 mimic similar curves found by other researchers, notably Treisman (1986, figure b, p112), for the analogous line target problem.

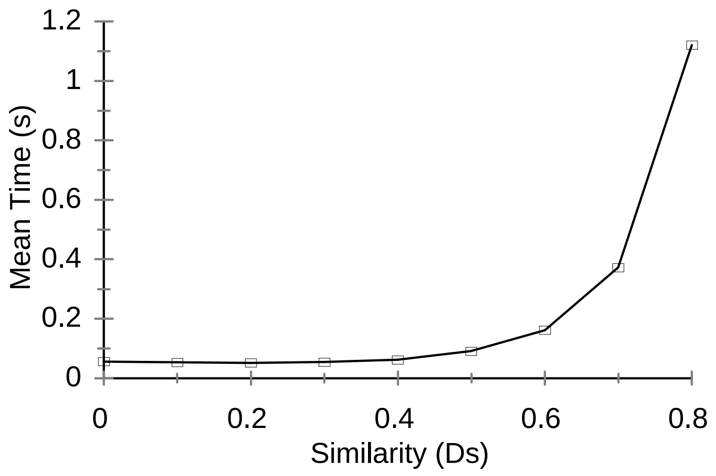

Figure 3-9: The effect of distractor similarity on mean convergence time for 32 distractors, a search space of size 5000 and a model of size 100.

Figure 3-9 shows how for a (large) constant number of distractors, 32, the mean convergence time increases in an exponential-like fashion as the similarity increases. Whether this curve mimics human data is difficult to determine from published graphs.

3-7 Experiment 6: The effect of Noise

The effect of noise in the model was determined by corrupting the model being added to the search space. The amount of noise, Qn, could be a number between zero and one indicating the proportion of elements to be corrupted, achieved in the program by adding one to those elements. The termination conditions also had to be altered as it would be unlikely that all agents would become active. Therefore, the three paramters, threshold, base threshold and stable time, were set as follows:

- threshold - set to equal the number of elements not corrupted by noise;
- base-threshold - set to be 50% of the Threshold;
- stable time - set at three.

These values are entirely arbitrary and, had SDS been unable to find the corrupted models with any degree of accuracy then they would have been altered. The next experiment takes a more detailed look at termination conditions. This experiment is concerned with relative performance for different noise levels, not overall performance.

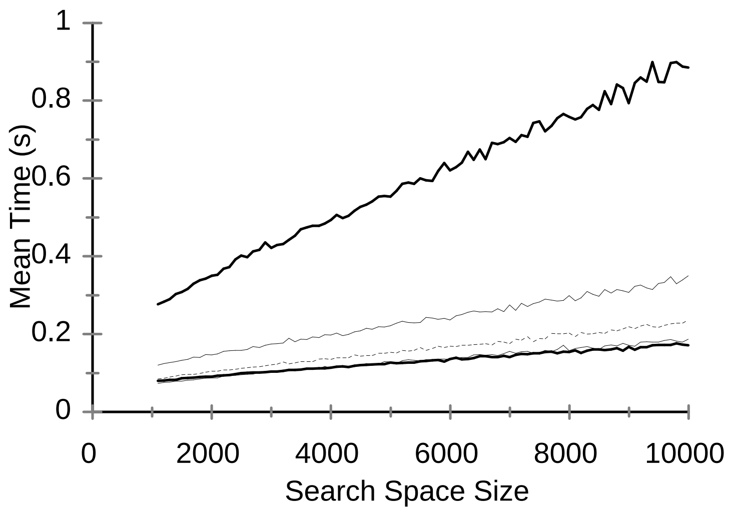

Figure 3-10: The effect of noise (Qn) on mean convergence time. Qn varies from 0, lowest line to 0.4 top line in steps of 0.1.

Figure 3-10 shows how for Qn≤0.2 there is little overall difference in mean convergence time. However, as Qn moves from 0.2 to 0.3 and through to 0.4 the increase in mean convergence time becomes larger. The performance for Qn=0.4 appears to be "bad", certainly in comparison with that of SDS for Qn≤3. Figure 3-11 shows its performance relative to an optimised sequential search algorithm (OptSeq, see Appendix 4). OptSeq searches through the search space and as soon as it reaches a match equal to that of the proportion of uncorrupted elements it terminates. At SS=3800 SDS outperforms

OptSeq. The mean convergence time of OptSeq approximately doubles each time the search space doubles whereas for SDS the mean convergence time increases by a factor of around 1.5. The OptSeq algorithm could be improved still further which would move the crossover point still further along the X axis. However, due to the nature of serial search and the level of noise in this experiment SDS will **always** eventually outperform sequential search because the mean convergence time will double every time the search space doubles in size.

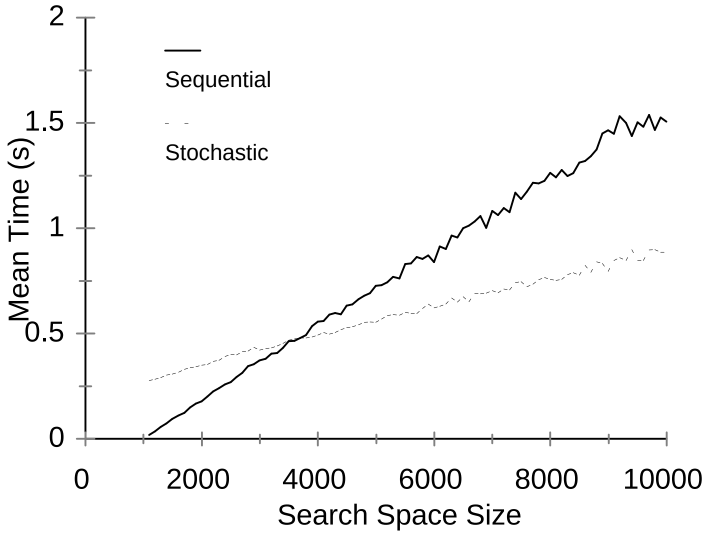

Figure 3-11: A comparison between optimised sequential search and SDS for Qn=0.4. The point at which SDS out performs the sequential search occurs at SS=3800.

3-8 Experiment 7: The Termination Conditions

The investigation of the termination conditions requires a completely different strategy. As outlined in chapter 2 there are three different termination parameters that can be varied, threshold (T), base threshold (BT) and stable time (ST).

Initially clean data will be used and the results from this might indicate how termination conditions are best applied to noisy data such that the mean convergence time can be lowered still further.

In order to be able to get some idea of algorithm behaviour with different termination conditions a search space that can contain a model of sufficient size is required.

However, there are time constraints so a model of 1000 and a search space of 10000 is not possible. Therefore, a search space of 5000 and a model of 500 was chosen.

After some consideration the strategy for determining the effect for the termination conditions was decided as:

- Set T=500, BT=500 and ST=1;
- Determine the lowest T for which error=0;
- For this T add 1 to ST and set BT to ½T;
- If error=0 decreases BT else increase BT or;
- If error persists increases ST.

The results of this strategy can be seen in Table 3-1.

T	ST	BT	Iterations	Time (s)	Error
500			24.51	0.0858	0.000
450			22.08	0.0805	0.000
400			20.82	0.0754	0.000
350			20.89	0.0758	0.000
300			20.74	0.0754	0.000
250			20.37	0.0735	0.000
200			20.05	0.0714	0.000
150			19.76	0.0711	0.000
100			18.78	0.0689	0.000
85			18.09	0.0666	0.000
80			18.04	0.0653	0.001
75			18.70	0.0678	0.005
50			2.32	0.0077	0.986
85	2	40	9.62	0.0358	0.584
	2	60	17.27	0.6440	0.051
	2	70	18.71	0.0681	0.001
	3	60	17.89	0.0665	0.024
	6	60	18.09	0.0666	0.001

Table 3-1: How altering the termination conditions affects the mean convergence time.

By simply reducing T, from 500 to 85, the mean convergence time has been reduced from 0.0858secs to 0.0666secs, a 23% performance increase. Increasing ST and BT did not necessarily decrease mean convergence time but introduced error. The error is the proportion of searches that resulted in an incorrect location of the model. Unfortunately, when noise is present, BT and ST are a requirement for convergence with minimal or no error. As can be seen the lowest mean convergence time with an error < 5% is when

T=85, ST=3 and BT=60. Increasing ST to 6 whilst reducing the error, which is still above zero, results in a mean convergence time equal to that when only the threshold was used. It seems that for clean data at least, only the threshold is important in terms of search time with no error.

Since the performance of SDS when Qn=0.4 deteriorated significantly from its performance with no noise, the data in table 3-1 should be useful in increasing its performance. A search space of 5000 and a model of 1000 were used so the results could be compared with those obtained in experiment 6. The results can be seen in table 3-2.

T	ST	BT	Iterations	Time(s)	Error
600	3	300	76.38	0.5535	0
400	4	200	64.99	0.4837	0
300	4	200	63.39	0.4760	0
200	3	150	61.18	0.4589	0
200	2	150	61.21	0.4573	0

Table 3-2: How the mean convergence time is affected by altering the termination conditions when noise is present.

The top line of data in table 3-2 shows the results from experiment in 7 for Qn=0.4. The other lines show how lowering T, ST and BT down to 200, 2 and 150 respectively has reduced the mean convergence time by almost 20%. The performance could probably be improved still further, but there appears to be no logical sequence as to how SDS will perform with different termination conditions. It is more a case of trial and error. Although once a set of parameters has been found that improves performance they can be systematically reduced or in the case of ST increased to reduce the mean convergence time still further.

3-9 Concluding Remarks

The algorithm depends on the size of the search space, but less so than might be expected. The time penalty for looking at larger search spaces is far less than with other search algorithms where a doubling in the search space would result in a doubling of the search time.

Up to a point, larger models are easier to find than smaller ones for a constant number of agents. However, if the number of agents is equal to the model size, the larger the model the longer SDS takes to find it.

The number of agents has a strong effect on search time. In the serial implementation there is a positive linear relationship between the number of agents and the mean search time. The more agents the greater the search time. However, in terms of the number of iterations, which is more important in evaluating the parallel nature of the algorithm, the search time actually decreases exponentially as more agents are added, until the number of agents is approximately equal to the size of the model.

A larger alphabet allows the algorithm to work more effectively. This is analogous to human vision where we see better in the day, not only because there is more light, but because we can see more colours and therefore it easier to distinguish items as distinct from each other.

The more similar a search space is to the model the harder it is for SDS to find the model. The results go a little way to showing one of the psychophysical characteristics of SDS, that is that when there are number of distractors which are similar enough to the model the search time increases as their number increases.

Unsurprisingly the greater the amount of noise in the model the longer SDS took to converge.   However, for $Qn \leq 0.2$ there was no significant difference between the mean convergence times. Furthermore, for search spaces larger than 3800 SDS outperformed an optimised sequential search. It is often the case that researchers will use the worst case algorithm to compare their best algorithms with. In this case although OptSeq is not a fully optimised sequential search algorithm no matter how optimised the algorithm is, it is a fact that as the search space doubles the search time will double also. Therefore, at some point SDS will outperform a sequential search because for this implementation of SDS the search time only increases by a factor of 1.4 every time the search space doubles.

Few conclusions can be drawn from the effect of the termination conditions as limiting time prevented a large investigation. A mathematical model, which has yet to be completed and is out of the scope of this project, would be necessary before any conclusions can be drawn. However, with suitable reductions in threshold, performance of SDS can be improved by up to, if not more than, 26% for clean data. Whereas for noisy data performance could be improved by up to 20% with suitable choices of base threshold and stability time.

Much more data needs to be collected to determine whether or not a self-terminating algorithm could be built where no information on noise is available to the algorithm. The data collected here does not allow for that kind of algorithm to be written.

Chapter 4

Application of SDS to Visual Search

4-1 Introduction

In order to apply SDS to human visual search tasks a two-dimensional implementation is required. Initially, the tasks it will be required to perform will involve the detection of oriented line targets. Generally such tasks are performed with single colour targets and distractors on a plain background. The implementation of 2D SDS will be written with this in mind.

4-2 Two-Dimensional SDS

The structure of the 2D SDS algorithm is identical to that of the 1D algorithm. However, the properties of the search space, the agents and the test phases are all different. The search space is two dimensional; the agents have to store co-ordinates rather than single figure positions. The test phase will also have to take the two dimensional nature of the algorithm into account. Only the diffusion phase remains the same in that each agent only tests against other active agents which then diffuse their mappings. The fact that the mappings will be in two dimensions should not alter the internal workings of the diffusion phase.

The search space will comprise 0's for the background and clusters of 1's for the targets and distractors (figure 4-2). The distractors are constructed using trigonometric formulae, ($x \sin \theta$, $y \cos \theta$), from their point of reference. They are constructed in such a way that no two adjacent elements on the same row are both zero. As long as the distractors orientation is $\leq 45^{O}$ this will not cause any breakage in the line unlike the situation shown in figure 4-1, where the 1's are not diagonally or vertically adjacent.

```
000
 010
   010
     010
       010
        000
```

Figure 4-1: The effect of distractor orientation >45^{O}.

Bishop (1988), however, notes that in one dimensional SDS targets such as this,

000000000111111111000000000,

where ‘1111111111', at position 10, is the target are not easily located accurately. The random nature of SDS means that the target could be located at any position between about 6 and 14. This is because if one or two active agents diffuse their mappings‘too quickly’ not enough of the search space will be sampled to determine that they are actually pointing to the wrong location.

```
00000000000000000000000000000000
00000000100000000100010000000000
00000000010000000010001000000000
00010000010000100010001000100000
00001000001000100001000100010000
00001000100000100100000010010000
00000100010000100010000001001000
00000000010000000010000001000000
00000000001000000001000000100000
00000000000000000000000000000000
```

Figure 4-2: An example of a small search used by 2D SDS. The target, four 1's in a vertical ‘line’ in the centre of the search space are surrounded by a number of ‘distractors’.

This could be a problem for 2D SDS when a vertical line is the target. One possible solution is for the algorithm to search for the target, and the border of 0's around it (figure 4-3). The algorithm would look for the location of the upper left zero of the model and determine its position from that. However, this did not entirely solve the problem of incorrect location. As fewer than half the elements in the target are 1's the algorithm was still not locating properly due to the number of active agents causing quick diffusion. The location errors were not just vertical translations of the actual model location, but errors in the horizontal as well.

```
000
010
010
010
010
000
```

Figure 4-3: The target and border.

The methodology by which an agent was labelled active or inactive during the test phase had to be altered radically. Currently, during the test phase, each agent only samples another position within the range of the target do determine its activity.

Another method, still using the border of zeros would check to see if its current location is a possible top left of the model by

- making sure its current position and that of both the positions to the right are both zero,
- making sure that the corresponding three elements Q+1 (the model length + 1) positions down from the current position contain zeros and
- that a random position between those two contains a zero-one-zero pattern.

Any agent that satisfies all of these criteria would be active otherwise it would be inactive.

```
00000000000000000000000000000000
00000000100000000100010000000000
00*000000100#0000100010000000000
00010000010000100010001000100000
00001000001000100001000100010000
00001000100000100100000010010000
00000100010000100010000001001000
00000000010000000010000001000000
00000000001000000001000000100000
00000000000000000000000000000000
```

Figure 4-4: A search space showing the current position of two agents, indicated by the * and #.

To clarify this method consider the position of agent * in figure 4-4, the search space value at this point is zero. This agent satisfies the first two conditions of activity, that the values of the next two rightwards positions are both zero and the corresponding positions Q+1 rows down are also zero. Its state of activity depends on the random choice of elements in between the previous two positions. If the agent chooses to sample the row immediately below its current position it will encounter a `010` and the agent will become active. However, should it choose any other of the possible positions it will not find a `010` and will be inactive. As can be seen, the agent `#` will always be active as the first two criteria of activity are both satisfied, and all the rows of three elements in between are of the `010` structure.

As a result of changing the activity criteria the algorithm worked, albeit painfully slowly. The results of an initial run can be seen in figure 4-5.

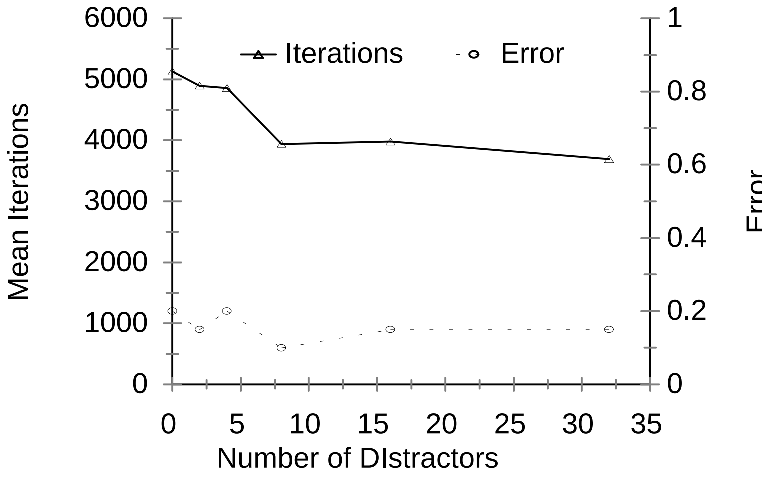

Figure 4-5: The mean convergence time (iterations) and error for 2D SDS as the number of distractors increases. The target is a vertical line, the distractors are oriented at 45^{O}. The search space is 1000 by 100 units and the target and distractors were of length 50 units. 200 agents were used.

These results show how the mean iterations to convergence fell as the number of distractors increased. An upper limit of 10000 iterations was set on the algorithm and the location errors made are due to the algorithm reaching this limit. The reason that the mean iterations decreases as the number of distractors increases is because the number of possibly active areas is, surprisingly, lower. When fewer distractors (that are as different from the target as these are) exist in the search space there are more areas of white space which, as far as the activity criteria are concerned to begin with, are possible candidates for an active agent. When the agents are more likely to point to a mapping that contains a 1, which is the case when the number of distractors increases, their overall activity level is lower and therefore incorrect mappings are less likely to be diffused. This is actually supported by research of human performance.

Bonneh and Sagi (1998) found that for large displays with only a few items on display, the proximity of the items to each other was important for aiding target detection. If the items are spread out each one has to be examined individually in a pseudo-serial manner. However when there are more items in the display the target is more likely to *pop out*. The method by which the search space in SDS is created means that for small numbers of distractors the items are widely spread out in the search space.

The current implementation of 2D SDS is a vertical filter. That is, it searches for vertical lines of a given length. The test phase would have to be altered to cope with different oriented targets. With more time this could be achieved. However, the only experiments that can be completed in the time available will use the vertical filter algorithm.

4-3 Applications of 2D SDS to human visual search tasks

The following two hypotheses concerning 2D SDS need to be tested:

1. The response of the SDS vertical filter to a display of vertical lines will be low. That is the overall level of activity will be spread out on the whole of the search space because there will be many different areas where the agents can be active. This would be indicated by relatively fast convergence times with active agents spread out amongst the whole search space. This would relate to Foster and Westland's (1995) research indicating the presence of two broadband filters, horizontal and vertical, in early visual processing.
2. For 'similar enough' distractors as the number of distractors increases the search time will increase. the algorithm will be tested for vertical line target detection with distractors at a number of different orientations. The orientations will range from 5^{O} to 25^{O} in steps of 5^{O}.

For these experiments the search space will be set at 1000x1000, the model length 50 and the number of agents 1000. 100 trials of each experiment will be conducted.

Ideally, the output of the 2D algorithm should be in the form of an activity map rather than the location of the target. This map would be initially zero at every point, then, at each iteration one would be added to the positions pointed to by active agents. This would give some idea as to the spread of activity in the search space over the length of the search. Unfortunately, analysing a large array (>256 in width) in most spreadsheets is not possible. Therefore inferences have to be made from the mean iterations to convergence.

4-4 Experiment 1: The response of 2D SDS to large numbers of the same item

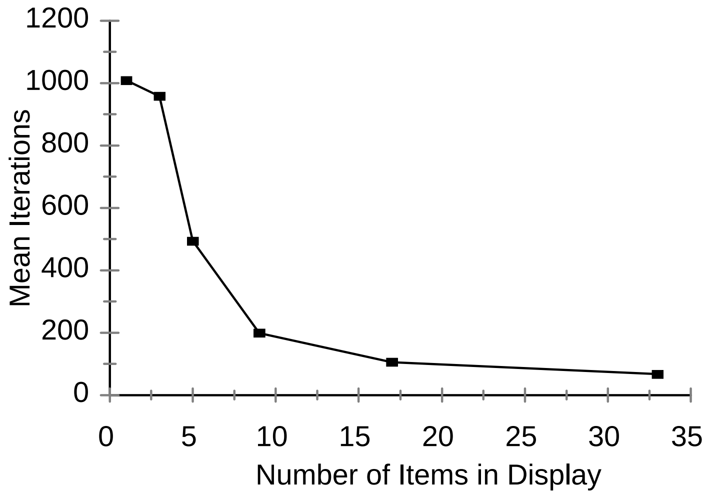

Figure 4-6: The mean number of iterations to convergence for a vertical filter SDS as the number of vertical items increases.

Figure 4-6 shows how the mean iterations decreases as the number of vertical items increases. This to be expected as there are far more active areas in a display with 33 items than there are with 1 item. It is likely, therefore, that all the agent's activity is spread over the whole search space. This would be considered as a low response to the search space and would indicate that if a target exists it is not a vertical line. If a horizontal filter 2D SDS were applied to the same search spaces its response would be low, but in a different way. It would only stop searching once it had reached the maximum iteration limit set in the algorithm as there are no areas that could feasiblely stay active for more than one or two iterations in a row. However, if a horizontal line was present, its response would be high, with a lot of activity in one particular area, indicating the presence of a target.

4-5 Experiment 2: The orientation discriminatory ability of 2D SDS

The results from this experiment are disappointing in that they do not mimic human data. Only one distractor orientation was tested, 5^{O}, because it was felt that with the results obtained it was not worth trying more experiments with different distractor orientations. Figure 4-7 shows how the search time stayed approximately the same as the number of distractors increased.

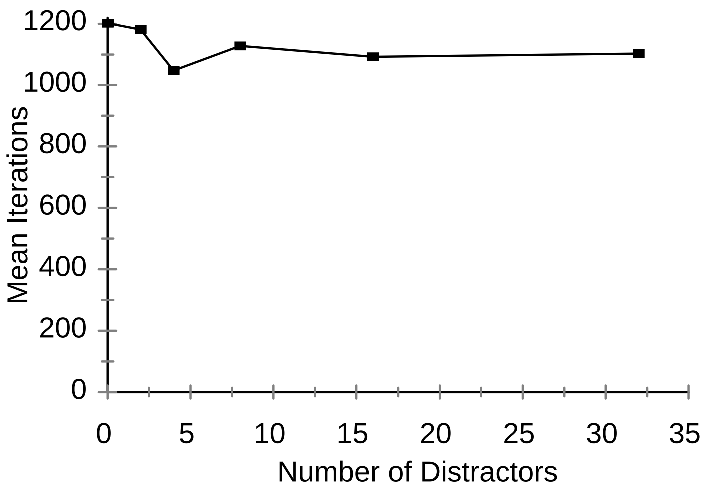

Figure 4-7: The effect of number of distractors on mean iterations to convergence for distractor orientation of 5^{O}.

The reasons for obtaining these results are unclear but it could be due to implementation, over simplification of the experiments, resolution of the search space or even that 2D SDS is not a good basis for a model of visual search. Whatever the reason, the results are a little disappointing. A significant drop or a significant increase in search time might have been expected.

Chapter 5

Discussion

The investigation of SDS in both its one and two-dimensional varieties has led to some interesting results.

In a pure parallel implementation the mean convergence time (iterations) of SDS is inversely proportional to the number of agents and is less dependent on the size of the model or search space. However, in its serial implementation the mean convergence time (seconds) is proportional to the number of agents but again has a lesser dependence on the size of the search space and model. The performance of SDS improves as the alphabet is increased. However, in the presence of noisy data, the algorithm begins to perform less and less efficiently, though with suitable tweaking of the termination conditions, performance can be improved by as much as 26% in some cases. As the search space increases in similarity to the model the performance decreases.

The limited set of results from the 2D model imply that the as the number of distractors in the search space increases the overall performance of SDS is unchanged. This occurred when the orientation of the distractors was only 5^{O}.

As the number of vertical items in the display increased it was not surprising that the vertical filter SDS converged more quickly.

Psychophysical plausibility

Some of these results, namely those obtained from the 1D distractor experiments, support the view that SDS has properties that are similar to those found in human psychophysical experiments. As stated before the 1D concept of distractors is analogous to the concept of oriented line target detection and the curves in figure 3-8 mimic the human data for such tasks. However there are some other results that support the psychophysical plausibility of SDS.

For a constant number of agents in a given search space size larger models are more easily found. This makes psychophysical sense. If SDS is similar to the algorithm used by the early visual system then it is likely that for each visual search performed the same number of agents are going to be used. Experiments carried out by Treisman and Gormican (1989) indicate that size is a feature and that larger items are more easily found.

Discrimination of objects by humans in a full colour world is much easier than in a black and white world. Efficient texture segmentation is thought to help with object location and recognition. Therefore, more complex textures can be generated the more colours that are available. SDS works better as the range of values each element in the search space can take increases.

In 2D SDS the results are less promising. However, initial results from the algorithm appear to correspond to predictions that can be drawn from Foster and Westland’s (1995) theory if assumptions about the resources of the early visual system are made; that the overall activity of a vertical filter when applied to a display containing vertical lines will be saturated across the whole visual field.

Unfortunately the distractor experiment did not support the view that SDS is psychophysically plausible. The performance of 2D SDS was almost unchanged as the number of distractors increased. At this distractor orientation the performance should decrease as the number of distractors increases.

Further Investigation

There are still a large number of areas that need to be investigated further. One of the most important aspects of SDS are the termination conditions, as these have been shown to alter the performance of the algorithm by up to 26%. A large amount of data needs to be collected and analysed to determine whether or not a pattern exists for calculating the best termination conditions for a given search space size, model size, number of agents and amount of noise. A self-terminating algorithm would be ideal and this could possibly be achieved with the help of an activity map. Thus, feedback from the search space could indicate to the algorithm the proportion of active agents the are with the greatest level of activity. This information could be used by the algorithm to determine its own termination criteria. For example, if the activity in one area of the search space was significantly higher than in the rest of the search space the algorithm would terminate even if the proportion of active agents was not particularly high. In particular termination of the 2D SDS algorithm requires investigation. No experiments were conducted in this area. The algorithm terminated when all the agents were active.

This implementation of 2D SDS is not based on any theoretical results. There is no implication that this implementation is in any way correct in a psychophysical sense. Perhaps groups of agents could be sent out in blocks to test the search space rather than single agents. In particular if the algorithm is to search for lines of any other orientation the test phase has to be rewritten.

The creation of the search space is also a problem. The resolution on the search space is very poor, as only vertical and 45^{O} lines are represented at all faithfully. Real human visual search tasks are not conducted with jagged edged lines. However, if aliasing was used to smooth out oriented lines then the whole nature of the test phase would have to be altered. The algorithm would need to be altered to cope with the extra variation in the range of values the search space could take. Would the algorithm look at intensity changes between pixels or would it still try to match values? This kind of situation will need to be investigated.

In order to obtain more information about the possible relationship between the current theories of human visual search and SDS the algorithm must output an activity map such as those found in the GS model. Although the creation of activity map is relatively simple its analysis might not be. Furthermore activity maps produced at each iteration would give a lot of insight into the characteristics of algorithm termination. These maps would show how, at first the algorithm randomly sends out agents in to the search space, before their progress becomes less random and they all converge to the same area.

Foster and Westland (1995) also produced a model of oriented-line-target detection with the two horizontal and vertical filters. The results they found concerning the presence of a horizontal and vertical filter in the very early visual system need to be examined in the context of 2D SDS. The application of 2D SDS to these experiments is important in discovering more about the characteristics of the algorithm.

Conclusions

In short SDS is an extremely interesting algorithm that has some psychophysical like properties. It has a pseudo-attentional mechanism by which the agents are guided to "interesting" areas of the search space. This corresponds well with the attentional spotlight present in the GS model.

However, the properties the algorithm exhibited in the one-dimensional form are not exhibited in its two-dimensional incarnation. This could be due to the creation of the search space, the implementation of the test phase or even the possibility that SDS is not a model for human visual attention.

No conclusions can be drawn either way as to its suitability as a model for attention. However, this research has raised some interesting questions that should certainly be answered.

Appendix 1

Simple Stochastic Diffusion Search

Purpose Built Header File - stoch.h

```
/* Mapping Cell Class for Stochastic Search Network */

class MC {
      public:
            MC();
            void setMap(int); // sets mapping of MC (mapping cell)
            void setFire(int); // firing status, 0 off, 1 on
            int Mapping(); // returns the mapping
            int Firing(); // returns the firing status

      private:
            int map, cFire;
            };

MC::MC() {cFire=0;}
void MC::setMap(int m) {map = m;}
void MC::setFire(int c) {cFire=c;};
int MC::Mapping() {return map;}
int MC::Firing() {return cFire;}
```

Main Program - sds.cpp

```
/* Simple Stochastic Search Network */
/* Outputs results to "results.dat" */
/* Robert Summers */
/* Version 3a, 20th June 1998 */

#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include<iostream.h>
#include<fstream.h>
```

```
#include<time.h>
#include<string.h>
#include"stoch.h"

int Q, numAgents, SS, origSS, posOfModel, numMax, corrupt,
numOfStrings;
int threshold, baseThreshold, stable, stabilityTime;
clock_t start,stop;
int iteration, numstr, hits, index, e, isSame, p;
int i,j,dummy,dummy2;
fstream g;
int maxSS, step, experiment;
char expNum[16];
char f1[]="results.dat";
MC *DMC;
int *model, *cmodel, *searchSpace; *element; *frequency;

void readInData();
void conditions();
void rndStr();
void init();
void test();
void diffuse();
int terminate();
void Xrandom();
int mode();

int main()
      {
      readInData();
      for (experiment=0;experiment<=(maxSS-origSS)/step;experiment++)
            {
            SS=origSS+experiment*step;
            itoa(experiment,expNum,10);
            strcat(expNum,f1);
            g.open(expNum, ios::out);
            g << "SS=" << SS << endl;
            g << "NumOfStrings=" << numOfStrings << endl;
            g << "Q=" << Q << endl;
            conditions();
            for (numstr=0;numstr<numOfStrings;numstr++)
                  {
                  g << numstr << "\t";
                  rndStr();
                  randomize();
                  start=clock();
                  init();
                  iteration=1;
                  test();
                  while ((terminate()==0) && (iteration < (2*SS)))
                        {
                        diffuse();
                        iteration++;
                        test();
                        }
                  stop=clock();
                  g << iteration << "\t" << ((float)
                                    ((stop-start)/CLK_TCK)) << "\t";
                  dummy=mode();
                  g << dummy << "\t" << posOfModel << "\t" <<
                                                 dummy-posOfModel;
                  cout << ".";
                  g << endl;
                  }
            g.close();
            }
            return 0;
      }

int mode()
      {
      int largest=0, modeValue=0;
      for (int rating=0; rating < SS; rating++) frequency[rating]=0;
      for (i=0;i<numAgents;i++)
            frequency[DMC[i].Mapping()]=frequency[DMC[i].Mapping()]+1;
      for (rating=0; rating < SS; rating++)
            {
            if (frequency[rating] > largest)
                  {
                  largest = frequency[rating];
                  modeValue = rating;
                  }
            }
      return modeValue;
      }
```

```
int terminate()
      {
      int terminate=0;
      if (hits>baseThreshold-1) {stabilityTime=stabilityTime+1;}
            else {stabilityTime=0;}
      if ((hits>threshold-1) || (stabilityTime==stable)) terminate=1;
      return terminate;
      }

void diffuse()
      {
      Xrandom();
      for (i=0;i<numAgents;i++)
            {
            if (DMC[i].Firing()==0)
                  {
                  p=element[i];
                  if (DMC[p].Firing()==1)
                        {
                        DMC[i].setMap(DMC[p].Mapping());
                        DMC[i].setFire(1);
                        }
                        else {DMC[i].setMap(rand() % SS);}
                  }
            }
      }

void test()
      {
      hits=0;
      Xrandom();
      for (i=0;i<numAgents;i++)
            {
            index=DMC[i].Mapping();
            e=element[i];
            if ((index+e)<SS)
                  {
                  if (model[e]==searchSpace[index+e])
                              {
                              DMC[i].setFire(1);
                              hits=hits+1;
                              }
                         else {DMC[i].setFire(0);}
                  }
                  else {DMC[i].setFire(0);}
            }
      }

void Xrandom()
      {
      int range;
      if (numAgents < Q) range=numAgents;
            else range=Q;
      for (i=0;i<numAgents;i++)element[i]=rand() % range;
      }


void init()
      {
      for (i=0;i<numAgents;i++)
            {
            DMC[i].setMap(rand() % SS);
            }
      }

void rndStr()
      {
      if (SS-Q==0) posOfModel=0; else posOfModel=rand()%(SS-Q);
      for (i=0;i<SS;i++) searchSpace[i]=rand()%numMax;
      for (i=posOfModel;i<posOfModel+Q;i++)
                        searchSpace[i]=cmodel[i-posOfModel];
      }
void conditions()
      {
      cout << "\nModel size " << Q << endl;
      cout << "Search Space Size " << SS << endl;
      DMC = (MC *) malloc(numAgents * sizeof(MC));
      element = (int *) malloc(numAgents * sizeof(int));
      searchSpace = (int *) malloc(SS * sizeof(int));
      frequency = (int *) malloc(SS * sizeof(int));
      g << "NumAgents=" << numAgents << endl;
      g << "SSNo.\tI\tTime\tMapping\tActual\tError" << endl;
      }


void readInData()
```

```
{
cout << "Enter the start size of the search space:";
cin >> origSS;
cout << "Enter the end size of the search space:";
cin >> maxSS;
cout << "Enter the increment:";
cin >> step;
cout << "Enter the size of the model:";
cin >> Q;
cout << "Enter the number of trials per experiment:";
cin >> numOfStrings;
cout << "Enter the alphabet of the search space:";
cin >> numMax;
cout << "Enter the number of corrupted data units:";
cin >> corrupt;
model = (int *) malloc(Q * sizeof(int));
cmodel = (int *) malloc(Q * sizeof(int));
srand(1);
for (i=0;i<Q;i++)
      {
      model[i]=rand()%numMax;
      cmodel[i]=model[i];
      }

if (corrupt>0) {
            for (i=0;i<Q;i=i+(int) (Q/corrupt))
                                    {cmodel[i]=cmodel[i]+1;}
            }
cout << "\nEnter the number of Agents required:";
cin >> numAgents;
cout << "\nEnter Threshold for termination:";
cin >> threshold;
cout << "\nEnter Stable time, (no. of iterations before program
                                              terminates ";
cout << "\nwhen no. of correct matches is above a base
                                              threshold:";
cin >> stable;
cout << "\nEnter Base Threshold for termination after " <<
                                    stable << " iterations:";
cin >> baseThreshold;
cout << endl;
}
```

Appendix 2

SDS with Distractors

Purpose built header file - stoch.h

See appendix 1.

Main Program - sdsdis.cpp

```
/* Stochastic Search Network with Distractors*/
/* Robert Summers */
/* Version 3a, 3rd June 1998 */

#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include<iostream.h>
#include<fstream.h>
#include<time.h>
#include"stoch.h"

int Q=100, numAgents=100, SS=5000, posOfModel, numMax, numDistract,
similarity, numOfStrings=1000;
int threshold, baseThreshold, stable, stabilityTime;
clock_t start,stop;
int iteration, numstr, hits, index, e, isSame, p;
int i,j,k,dummy;
float dummy2;
fstream g;

MC *DMC;
int *model, *cmodel, *searchSpace; *element; *frequency; // model,
corrupt model, etc.

void readInData();
void conditions();
void rndStr();
void init();
void test();
void diffuse();
int terminate();
void Xrandom();
int mode();

int main()
      {
      randomize();
      readInData();
      conditions();
      for (numstr=0;numstr<numOfStrings;numstr++)
            {
            g << numstr << ";";
            rndStr();
            start=clock();
            init();
            iteration=1;
            test();
            while ((terminate()==0) && (iteration < (2*SS)))
                  {
                  diffuse();
                  iteration++;
                  test();
                  }
            stop=clock();
            g<<iteration<<";"<<((float)((stop-start)/CLK_TCK)) << ";";
            dummy=mode();
            g << dummy << ";" << posOfModel << ";" <<
                                                    dummy-posOfModel;
            cout << ".";
            g << endl;
            }
      g.close();
      return 0;
      }

int mode()
      {
      int largest=0, modeValue=0;
```

```
            for (int rating=0; rating < SS; rating++) frequency[rating]=0;
            for (i=0;i<numAgents;i++)
                  frequency[DMC[i].Mapping()]=frequency[DMC[i].Mapping()]+1;
            for (rating=0; rating < SS; rating++)
                  {
                  if (frequency[rating] > largest)
                        {
                        largest = frequency[rating];
                        modeValue = rating;
                        }
                  }
            return modeValue;
            }

      int terminate()
            {
            int terminate=0;
            if (hits>baseThreshold-1) {stabilityTime=stabilityTime+1;}
                  else {stabilityTime=0;}
            if ((hits>threshold-1) || (stabilityTime==stable)) terminate=1;
            return terminate;
            }

      void diffuse()
            {
            Xrandom();
            for (i=0;i<numAgents;i++)
                  {
                  if (DMC[i].Firing()==0)
                        {
                        p=element[i];
                        if (DMC[p].Firing()==1)
                              {
                              DMC[i].setMap(DMC[p].Mapping());
                              DMC[i].setFire(1);
                              }
                              else {DMC[i].setMap(rand() % SS);}
                        }
                  }
            }

      void test()
            {
```

```
      hits=0;
      Xrandom();
      for (i=0;i<numAgents;i++)
            {
            index=DMC[i].Mapping();
            e=element[i];
            if ((index+e)<SS)
                  {
                  if (model[e]==searchSpace[index+e])
                              {
                              DMC[i].setFire(1);
                              hits=hits+1;
                              }
                         else {DMC[i].setFire(0);}
                  }
                  else {DMC[i].setFire(0);}
            }
      }

void Xrandom()
      {
      int range;
      if (numAgents < Q) range=numAgents;
            else range=Q;
      for (i=0;i<numAgents;i++)element[i]=rand() % range;
      }


void init()
      {
      for (i=0;i<numAgents;i++)
            {
            DMC[i].setMap(rand() % SS);
            }
      }

void rndStr()
      {
      dummy2=(SS-Q)/numDistract;
      for (i=0;i<SS;i++) searchSpace[i]=0;
      posOfModel=rand()%(SS-Q);
      for (i=0;i<numDistract;i++)
            {
```

```
            j=dummy2*i;
            for (k=0;k<similarity;k++) searchSpace[k+j]=cmodel[k];
            }
      for (i=0;i<Q;i++) searchSpace[i+posOfModel]=model[i];
      }

void conditions()
      {
      cout << "Model size " << Q << endl;
      cout << "Search Space Size " << SS << endl;
      threshold=numAgents;
      stable=2;
      baseThreshold=numAgents;
      cout << endl;
      DMC = (MC *) malloc(numAgents * sizeof(MC));
      element = (int *) malloc(numAgents * sizeof(int));
      g << "NumAgents=" << numAgents << endl;
      g << "SSNo.;I;Time;Mapping;Actual;Error;" << endl;
      }

void readInData()
      {
      g.open("results.dat", ios::out);
      cout << "\nSize of the search space:";
      cout << SS;
      cout << "\nSize of the model:";
      cout << Q;
      cout <<"\nEnter number of distractors in the Search Space:";
      cin >> numDistract;
      cout << "Enter the similiarity of the distractors to the model
(0-1):";
      cin >> dummy2;
      similarity=(int) (dummy2*Q);
      numOfStrings=1000;
      numMax=10;
      g << "SS=" << SS << endl;
      g << "Similarity=" << dummy2 << endl;
      g << "numDis=" << numDistract << endl;
      model = (int *) malloc(Q * sizeof(int));
      cmodel = (int *) malloc(similarity * sizeof(int));
      searchSpace = (int *) malloc(SS * sizeof(int));
      frequency = (int *) malloc(SS * sizeof(int));
      for (i=0;i<Q;i++)
            {model[i]=rand()%numMax;}
      for (i=0;i<similarity;i++)
            {cmodel[i]=model[i];}
      }
```

Appendix 3

2D SDS

Purpose Built Header File - 2Dstoch.h

```
// 2 Dimensional Mapping Cell/Agent Header for 2dsds
class MC {
      public:
            MC();
            void setMap(int, int);
            void setFire(int);
            int MappingX();
            int MappingY();
            int Firing();

      private:
            int mapX, mapY, cFire;
            };

MC::MC() {cFire=0;}
void MC::setMap(int x, int y) {mapX=x; mapY=y;}
void MC::setFire(int c) {cFire=c;}
int MC::MappingX() {return mapX;}
int MC::MappingY() {return mapY;}
int MC::Firing() {return cFire;}

class Model {
      public:
            Model();
            void setLength(int);
            int getLength();
            void setPos(int, int);
            int modelX();
            int modelY();
      private:
            int length, X, Y;
            };

Model::Model() {length=0;}
void Model::setLength(int l) {length=l;}
int Model::getLength() {return length;}
void Model::setPos(int x, int y) {X=x;Y=y;};
int Model::modelX() {return X;}
int Model::modelY() {return Y;}

class Distractor {
      public:
            Distractor();
            void setLength(int);
            void setOrientation(int);
            int getLength();
            int getOrient();
            void setNumDis(int);
            int getNumDis();

      private:
            int length, orient, numDis;
            };

Distractor::Distractor() {length=0; orient=0;}
void Distractor::setLength(int l) {length=l;}
void Distractor::setOrientation(int o) {orient=o;}
int Distractor::getLength() {return length;}
int Distractor::getOrient() {return orient;}
void Distractor::setNumDis(int n) {numDis=n;}
int Distractor::getNumDis() {return numDis;}
```

Main Program - 2Dsds.cpp

```
/* 2D SDS */
/* Robert Summers */
/* Version 0.1 , 29th July 1998 */

#include<stdlib.h>
#include<math.h>
#include<stdio.h>
```

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include"2dstoch.h"

void createSpace(Model, Distractor);
void init();
int test();
void diffuse();
void mode();

const float pi=3.14159265359;
const int sizeX=1000;
const int sizeY=1000;

int huge ss[sizeX][sizeY];
int huge frequency[sizeX][sizeY];

Model Q;
Distractor D;
MC *DMC;
int numAgents, modelPosX, modelPosY;

fstream g;
char expNum[14];
int numTrials, experiment;


void main()
      {
      randomize();
      int hits, iteration,trial,numDis;
      int dummy;
      cout << "Search Space Size:" << sizeX << "," << sizeY <<endl;
      cout << "Enter Distractor Orientation (degrees):";
      cin >> dummy;
      D.setOrientation(dummy);
      cout << "Enter model length:";
      cin >> dummy;
      Q.setLength(dummy);

      cout << "Distractors 0,2,4,8,16,32\n";
      cout << "Enter number of trials per experiment:";
```

```
cin >> numTrials;


for (experiment=0;experiment<6;experiment++)
      {
      if (experiment==0) {numDis=0;} else {numDis=(int)
                                    pow(2,experiment);}

      D.setNumDis(numDis);
      numAgents=1000; // =(Q.getLength()+2);
      D.setLength(Q.getLength());
      g.close();
      if (experiment==0) {g.open("0results.dat",ios::out);}
      if (experiment==1) {g.open("1results.dat",ios::out);}
      if (experiment==2) {g.open("2results.dat",ios::out);}
      if (experiment==3) {g.open("3results.dat",ios::out);}
      if (experiment==4) {g.open("4results.dat",ios::out);}
      if (experiment==5) {g.open("5results.dat",ios::out);}

      cout << "Number of Distractors:" << D.getNumDis() << endl;
      cout << "Number of Agents:" << numAgents << endl;

      g << "Orient:\t" << D.getOrient() << endl;
      g << "Length:\t" << Q.getLength() << endl;
      g << "Number of Distractors:\t" << D.getNumDis() << endl;
      g << "SS Size\t" << sizeX << "\t" << sizeY << endl;

      g << "Iterations\tXerror\tYerror\tError" << endl;

      for (trial=0;trial<numTrials;trial++)
            {
            Q.setPos(rand()%(sizeX-2)+1,
                              rand()%(sizeY-Q.getLength()-1)+1);
            DMC = (MC *) malloc(numAgents * sizeof(MC));
            createSpace(Q, D);
            init();
            iteration=1;
            hits=test();
            while ((hits < numAgents-1) && (iteration < 10000))
                  {
                  diffuse();
                  iteration++;
                  hits=test();
```

```
                        }
                  cout << ".";
                  mode();
                  g << iteration << "\t" << modelPosX-Q.modelX() <<
                              "\t" << modelPosY-Q.modelY() << "\t" <<
                              abs(modelPosX-Q.modelX()) +
                              abs(modelPosY-Q.modelY()) << endl;
                  }
            cout << endl;
            g.close();
            }
      }//end main();

void mode()
      {
      int largest=0;
      int i,j;
      for (j=0;j<sizeY;j++)
            {
            for (i=0;i<sizeX;i++)
                  {
                  frequency[i][j]=0;
                  }
            }
      for (i=0; i<numAgents; i++)
            {
            frequency[DMC[i].MappingX()][DMC[i].MappingY()]++;
            }
      for (j=0;j<sizeY;j++)
            {
            for (i=0;i<sizeX;i++)
                  {
                  if (frequency[i][j]>largest)
                        {
                        largest=frequency[i][j];
                        modelPosX=i+1;
                        modelPosY=j+1;
                        }
                  }
            }
      }

void diffuse()
```

```
        {
        int p;
        for (int i=0;i<numAgents;i++)
              {
              if (DMC[i].Firing()==0)
                    {
                    p=rand()%numAgents;
                    if (DMC[p].Firing()==1)
                          {
                          DMC[i].setMap(DMC[p].MappingX(),
                                                  DMC[p].MappingY());
                          DMC[i].setFire(1);
                          }
                          else {DMC[i].setMap(rand()%(sizeX-3),
                                      rand()%(sizeY-Q.getLength()-2));}
                    }
              }
        }


  int test()
        {
        int h=0;
        int indexX, indexY, elementY, possibleActive;
        int temp[3]={0,1,0};
        for (int i=0;i<numAgents;i++)
              {
              possibleActive=0;
              indexX=DMC[i].MappingX();
              indexY=DMC[i].MappingY();
              elementY=1+rand()%Q.getLength();
              if (ss[indexX+1][indexY]==0) {possibleActive++;}
              if (ss[indexX+1][indexY+Q.getLength()+1]==0)
                                                  {possibleActive++;}
              for (int j=0;j<3;j++)
                    {
                    if (ss[indexX+j][indexY+elementY]==temp[j])
                                                  {possibleActive++;}
                    }

              if (possibleActive==5)
                                {
                                DMC[i].setFire(1);
                                h++;
                                }
                        else {DMC[i].setFire(0);}
                }
        return h;
        }

void init()
        {
        for (int i=0;i<numAgents;i++)
                {
        DMC[i].setMap(rand()%(sizeX-3),rand()%(sizeY-Q.getLength()-2));
                }
        }

void createSpace(Model mm, Distractor dd)
        {
        int i,j,k;
        int length, numDis, posX, posY;
        float orient;
        int XnumDis, YnumDis;
        length=mm.getLength();
        orient=(pi/180) * dd.getOrient();
        numDis=dd.getNumDis();
        posX=mm.modelX();
        posY=mm.modelY();
        for (j=0;j<sizeY;j++)
                {
                for (i=0;i<sizeX;i++)
                        {
                        ss[i][j]=0;
                        }
                }
        XnumDis=(int) ((sizeX * numDis)/(sizeX+sizeY));
        if (numDis>0) YnumDis=numDis/XnumDis;
        int dumX, dumY;
        for (j=0;j<YnumDis;j++)
                {
                for (i=0;i<XnumDis;i++)
                        {
                        dumY=(int) (j*(sizeY)/YnumDis);
                        dumX=(int) (i*(sizeX-(length*sin(orient)))/XnumDis);
                        for (k=0;k<length;k++)
```

```
                  {
                  ss[dumX+(int) (k*sin(orient))][dumY+(int)
                                  (k*cos(orient))]=1;
                  }
            }
      }
ss[posX-1][posY-1]=0;
ss[posX][posY-1]=0;
ss[posX+1][posY-1]=0;
for (k=0;k<length;k++)
      {
      ss[posX-1][posY+k]=0;
      ss[posX][posY+k]=1;
      ss[posX+1][posY+k]=0;
      }
ss[posX-1][posY+k+1]=0;
ss[posX][posY+k+1]=0;
ss[posX+1][posY+k+1]=0;
}
```

Appendix 4

Optimised Sequential Search

```
/* Optimised Sequential Search Algorithm outputs to "results.seq" */
/* Robert Summers */
/* Version 2, 3rd June 1998 */

#include<stdlib.h>
#include<stdio.h>
#include<iostream.h>
#include<fstream.h>
#include<math.h>
#include<time.h>
#include<string.h>

int Q, SS, posOfModel, numMax, corrupt;
int origSS,maxSS,step,experiment;
int *model, *cmodel;
int *searchSpace;
int iteration,i,j,hits,numstr,numOfStrings,maxHits,maxPos, dummy2;
clock_t start, stop;
char expNum[16];
char f1[]="results.seq";
fstream g;

void readInData();
void rndStr();
void test();
int terminate();


int main()
      {
      cout << "optimised sequential search, requires 100% match
                                  between target and model\n\n\n";
      readInData();
```

```
        for (experiment=0;experiment<=(maxSS-origSS)/step;experiment++)
              {
              SS=origSS+experiment*step;
              itoa(experiment,expNum,10);
              strcat(expNum,f1);
              g.open(expNum, ios::out);
              cout << "\n\nSS=" << SS << endl;
              g << "SS=" << SS << endl;
              g << "NumOfStrings=" << numOfStrings << endl;
              g << "Q=" << Q << endl;
              searchSpace = (int *) malloc(SS * sizeof(int));
              g << "SSNo.\tIterations\tTime\tMapping\tActual\tError" <<
                                                              endl;\
              for (numstr=0;numstr<numOfStrings;numstr++)
                    {
                    maxPos=0;
                    maxHits=0;
                    rndStr();
                    start=clock();
                    hits=0;
                    iteration=0;
                    while ((terminate()==0) && (iteration<(SS-Q)))
                          {
                          test();
                          iteration++;
                          }
                    stop=clock();
                    g << numstr << "\t" << iteration << "\t" <<
                        (stop-start)/CLK_TCK << "\t" << maxPos << "\t";
                    g << posOfModel << "\t" << maxPos-posOfModel<< endl;
                    cout << ".";
                    }
              g.close();
              }
        return 0;
        }

  void test()
        {
        hits=0;
        if (i+iteration<SS)
              {
              i=0;
```

```
                        while ((i<Q) && (hits==i))
                              {
                              if (model[i]-searchSpace[i+iteration]==0) hits++;
                              i++;
                              }
                        }
                   }

            int terminate()
                  {
                  int terminate=0;
                  if ((iteration>0) && (hits>maxHits))
                        {
                        maxHits=hits;
                        maxPos=iteration-1;
                        }
                  if (maxHits==Q) {terminate=1;}
                  return terminate;
                  }

            void readInData()
                  {
                  cout << "Enter the start size of the search space:";
                  cin >> origSS;
                  cout << "Enter the end size of the search space:";
                  cin >> maxSS;
                  cout << "Enter the increment value:";
                  cin >> step;
                  cout << "Enter the size of the model:";
                  cin >> Q;
                  cout << "Enter the number of strings:";
                  cin >> numOfStrings;
                  cout << "Enter the alphabet:";
                  cin >> numMax;
                  cout << "Enter the number of corrupted data units:";
                  cin >> corrupt;
                  model = (int *) malloc(Q * sizeof(int));
                  cmodel = (int *) malloc(Q * sizeof(int));
                  srand(1);
                  for (i=0;i<Q;i++)
                        {
                        model[i]=rand()%numMax;
                        cmodel[i]=model[i];
                        }
                  for (i=0;i<corrupt;i++)
                        {
                  dummy2=(int) (i*Q/corrupt);
                  cmodel[dummy2]=(cmodel[dummy2]+rand()%(numMax-2)+1)%numMax;
                        }
                  }

            void rndStr()
                  {
                  posOfModel=rand()%(SS-Q);
                  for (i=0;i<posOfModel;i++) searchSpace[i]=rand()%numMax;
                  for (i=posOfModel;i<posOfModel+Q;i++)
                                    searchSpace[i]=cmodel[i-posOfModel];
                  for (i=posOfModel+Q;i<SS;i++)searchSpace[i]=rand()%numMax;
                  i=0;
                  }
```

References and Bibliography

Bishop, J. M., 1989, **Stochastic Searching Networks**, Proc. 1st IEE Conference on Artificial Neural Networks, pp329-331, London.

Bishop, J. M., 1989, **Anarchic Techniques for Pattern Classification**, PhD Thesis, Reading University, UK.

Bishop, J. M. and Torr, P., 1992, **The Stochastic Search Network**, Chapter 18 in Linggard, R., Myers, D. J. and Nightingale, C. (eds), **Neural Networks for Vision, Speech and Natural Language**., Chapman and Hall, London.

Bonneh, Y. and Sagi, D., 1998, **Effects of Spatial Configuration on Contrast Detection.** Visual Research, 38(22), pp3541-3553.

Chun, M. M. and Wolfe, J. M., 1996, **Just Say No: How are Visual Searches Terminated When There is No Target Present?** Cognitive Psychology, 30, pp39-78.

Gerrissen, J. F., 1991, **On the Network-based Emulation of Human Visual Search.** Neural Networks, 4, pp. 543-564.

Green, M., 1991, **Visual search, visual streams and visual architectures**. Perception and Psychophysics, 50(4), pp388-403.

Hubel, D. H., and Wiesel, T. N., 1968. **Receptive fields and functional architecture of monkey striate cortex**. Journal of Physiology, 195, pp215-243.

Marr, D., 1982 , **Vision.** Freeman, New York.

Treisman, A., and Gelade, G., 1980, **A Feature-Integration Theory of Attention.** Cognitive Psychology, 12, pp97-136.

Treisman, A., 1986, **Features and Objects in Visual Processing**. Scientific American, 225.

Treisman, A., 1988, **Features and Objects: The Fourteenth Bartlett Memorial Lecture.** The Quarterly Journal of Experimental Psychology, 40A(2), pp201-237.

Treisman, A., and Sato, S., 1990, **Conjunction search revisited**. Journal of Experimental Psychology: Human Perception and Performance, 16(3), pp459-478.

Treisman, A., 1993. **The perception of features and objects.** Chapter 1, in A. Baddeley & L. Weiskrantz (Eds.), **Attention: Selection, awareness, and control.**, (pp. 5-35). Oxford: Clarendon Press.

Westland, S. and Foster, D. H., 1995, **Optimized model of oriented-line-target detection using vertical and horizontal filters.** Journal of the Optical Society of America, 12(8), pp1617-1622.

Wolfe, J. M., Cave, K. C. and Franzel, S. L., 1989, **Guided Search: An alternative to the Feature Integration model for Visual Search.** Journal of Experimental Psychology: Human Perception and Performance, 15(3), pp419-433.

Wolfe, J. M., 1994, **Guided Search 2.0: A revised model of visual search.** Psychonomic Bulletin and Review, 1(2), pp202-238.

Wolfe, J.M. and Gancarz, G., 1997, **Guided Search 3.0**, WolfeLab; http://www.dahlen.com/kari/gs3.html

Wolfe, J.M., 1996, **Visual Search: A review,** in Pashler, H. (Ed.), **Attention**, University College London Press; at http://www.dahlen.com/kari/reviewcontents.html.

Zenger, B. and Fahle, M. 1997, **Missed Targets are More Frequent than False Alarms: A Model for Error Rates in Visual Search.** Journal Of Experimental Psychology: Human Perception and Performance, 23(6), pp1783-1791.