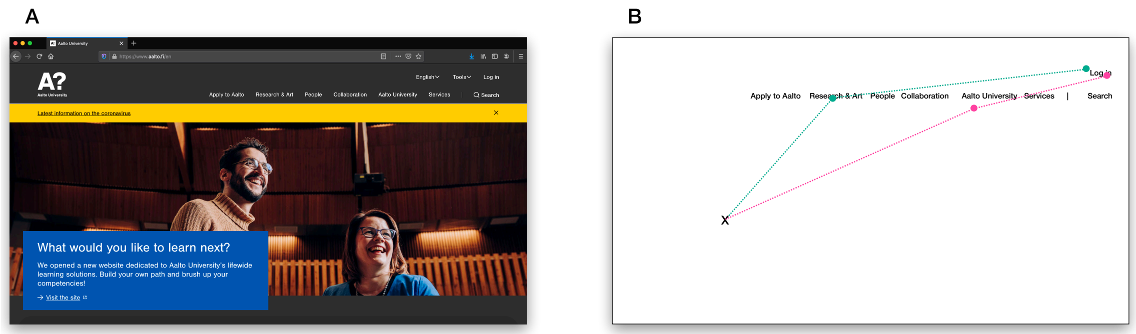
Identifying Graphical Layouts where Different Visual Search Strategies Emerge for Myopic and Planning Observers

AINI PUTKONEN



What would you like to learn next?

We opened a new website dedicated to Aalto University's lifewide learning solutions. Build your own path and brush up your competencies!

Fig. 1. Visual search task, where a user needs to find a "Log in" button on Aalto landing page (May 2021). A. The graphical layout. B. Two potential visual search strategies, when the user starts at X.

The ability of the human visual system to find a target on a user interface is critical for several tasks people perform on interactive systems. These visual search tasks include inspecting an unknown graphical layout to detect elements key to performing a task, e.g. buttons or input fields. Most investigated approaches in visual search tasks are myopic, only maximising the immediate reward from the following eye-movement. However, planning gaze sequences beyond the following action may be beneficial in tasks performed on new graphical layouts, as this allows collecting additional information about the environment. Previous work has suggested that people are, indeed, able to plan gaze sequences, but this finding has only been validated in abstract environments, namely non-descriptive shapes. In order to embed understanding of sequential planning of gaze sequences in interactive technologies, a better understanding of this behaviour in realistic environments is required. This paper presents an approach of framing a visual search in a realistic graphical layout as a sequential decision-making task modelled as a Partially Observable Markov Decision Process. Two observers distinguished by their discount rate, *myopic* and *planning*, are compared in visual search tasks where targets are located on a graphical layout. The results suggest that further investigation is required in identifying conditions under which graphical layouts lead to differing behaviour between the two observers. We provide suggestions on how the computational model should be extended. Finally, we propose how this approach can be used in investigating potential eye-movement strategies in real graphical layouts.

CCS Concepts: • **Human-centered computing** → **Empirical studies in HCI**; **User models**.

Additional Key Words and Phrases: visual search, sequential planning, POMDP, user modelling

ACM Reference Format:
Aini Putkonen. 2021. Identifying Graphical Layouts where Different Visual Search Strategies Emerge for Myopic and Planning Observers. In *Engineering For Humans - ELEC-E7861.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Detecting a target element among distractors is a frequent action performed on different user interfaces. When this detection takes place using the human visual system, the task can be described as *visual search*, broadly defined as locating target items in an environment [1]. For instance, a user who wishes to login to the Aalto University website first has to detect the "Log in" button, the target, on the landing page among other UI elements, the distractors (Figure 1). To perform this task efficiently, a user may choose between different strategies. Many investigated strategies aiming to explain behaviour in these visual tasks are myopic, only taking into account the reward from the immediately following action (e.g. [5, 8]). However, recently Hoppe and Rothkopf [4] have suggested that the human visual system is capable of *planning* beyond the following action, but this finding has not been validated in graphical layouts. In order to embed this understanding of sequential decision-making to real interactive systems, better understanding of human visual search in these environments is required.

This paper presents a computational model for detecting graphical layouts where different strategies emerge for myopic and planning observers. We frame the visual search task as a Partially Observable Markov Decision Process (POMDP), following approaches presented in [4, 7]. The degree to which the two observers take into account future rewards is determined by the discount factor γ in the POMDP, i.e. a myopic observer would have a γ close to 0, whereas a planning observer would have a γ close to 1. This allows us to compare behaviour of the agents arising in different graphical layouts.

POMDP approaches have been successful in explaining human behaviour in various contexts including the human visual system, e.g. in driving [7] and gaze-based selection [1, 3]. However, using POMDPs to analyse realistic visual search environments where different behaviours emerge could be further investigated. Hence, we extend the idea proposed by Hoppe and Rothkopf [4] from studying abstract shapes to a case where the stimulus is a graphical layout. The experimental setup Hoppe and Rothkopf use does not map well to tasks users face in their day-to-day interactions with technology. Hence, the ecological validity of the model needs to be further investigated in more naturalistic settings, including those taking into account full peripheral vision.

The computational model presented in this paper has two practical implications. Firstly, the identified layouts can be used in an empirical study to validate whether human visual system follows a sequential planning strategy in realistic graphical layouts. Secondly, the model can be used to analyse existing graphical layouts to identify strategies that emerge in them. Hence, this paper makes the following contributions:

- We propose a computational model formulated as a POMDP to identify graphical layouts where differing behaviour emerges for a myopic and a planning observer.
- We suggest how this computational model can be used to analyse existing graphical layouts.

The rest of this paper is organised as follows. A brief literature review of related work is presented in section 2. Section 3 describes the visual search problem and the POMDP formulation. Results are presented in section 5, followed by discussion in section 6.

2 RELATED WORK

Here we present a brief overview of literature covering eye-movement strategies. Models of human visual search aim to capture how primates direct their gaze from one location to another, behaviour arising from the serial nature of the human visual system. Due to limited capability of the brain to process information provided by the optic nerve, areas of the visual input are processed serially, as opposed to in parallel [5]. Several computational models have been proposed to explain the strategy humans employ in choosing the next location to fixate on, including approaches relying on heuristics, optimal control and sequential planning.

2.1 Heuristic approaches

Saliency maps are an example of a heuristic approach, where focus is directed to the area with highest saliency. Itti and Koch [5] suggest one such approach drawing from bottom-up control of attention, where the attributes of the visual stimulus direct selective attention. Their model generates a two-dimensional map encoding the environment using visual saliency – representation of how well objects stand out from their environment. In this model, gaze is directed to the target with highest visual saliency using a winner-take-it-all algorithm. The drawbacks of saliency-based approaches are threefold: (i) they assume that the process of choosing next location to gaze is myopic, only taking into account the following eye movement, (ii) information accumulated over time is disregarded and (iii) the bounds of the visual system are ignored [2]. Another heuristic approach is presented by Najemnik and Geisler [9], building on their previous work [8].Their original model of an 'ideal searcher' uses statistical information about the visual environment to direct gaze to location that maximises information gain. However, as this Bayesian computation is fairly complex, Najemnik and Geisler [9] propose a simple entropy limit minimisation heuristic, which produces behaviour similar to observed in humans. That said, they recognise that the heuristic is still somewhat implausible when compared to biological human visual system.

2.2 Optimal control approaches

Optimal control approaches aim to maximise the utility of a task, relying on limits imposed by the human cognition. Butko and Movellan [2] propose approaching the visual search task using a POMDP to capture the charactersitics of the visual system, building on the model proposed by Najemnik and Geisler [8]. Their approach includes the observation that sometimes it may be beneficial for an observer to fixate on less probable target locations to gain information. Incorporating this idea of computational rationality in visual search tasks has been also extended to capture more complex behaviour, including studies with realistic graphical layouts and learning. Jokinen et al. [6] suggest an approach capturing the transition of a user from a novice to an expert, using realistic graphical layouts. This model assumes that the visual system fixates on an item maximising expected utility. The calculation of this utility depends on level of experience of the user: first it is estimated by unguided perception, but it adapts to take into account long-term memory when the user learns the layout. Whereas these studies can explain eye movements observed in human data, they focus less on explaining situations where sequential planning may arise.

2.3 Sequential planning approaches

Sequential planning approaches investigate whether the human visual system is capable of planning beyond the next action. Hoppe and Rothkopf [4] present a comparison of myopic and planning approaches in two cases: one where only one saccade is possible and another where two saccades are possible. Their findings suggest that the human visual system is capable of planning beyond the next action. However, the ecological validity of this study can be challenged, as the visual search tasks was performed on abstract shapes. Addressing when sequential planning is useful is still widely unexplored in the literature concerning visual search tasks, especially in realistic environments.

3 PROBLEM FORMULATION

RThis section describes the visual search task used in this paper. In addition , the POMDP architecture for the models is briefly described.

3.1 Visual search task

We consider a visual search task, where an agent is trying to detect targets T among distractors D. Broadly, almost any graphical layout can be described as such a task, where there is one target element (e.g. a "Log in" button) among various distractors (e.g. "Sign up" button or input fields). We consider a simplified version of such layout, where the number of distractors is limited, as the environment in the visual search task. We hypothesise that different strategies emerge as a response to the properties of this environment, e.g. the location of the targets or screen size. Take a case where there are two elements (T and D) on the screen. The user needs to locate the target on the screen as quickly as possible, as often is the case in web search tasks. If the user only given time to complete one fixation, the best strategy is to gaze directly at the most likely target. However, if the user is given time for two fixations, there are two possible strategies (see Figure 2A). The user starts at location (x_0, y_0). They may either 1) go to D first, believing this is the most likely target, then moving to T (π_{myopic}) or 2) move between the targets to collect more information with peripheral vision and then go to the target that's the most likely one ($\pi_{planned}$). The layout can be thought as a grid, where the user can fixate on any of the rectangles $a^{i,j}$ at a given point in time (see Figure 2B). This grid representation is used in the POMDP formulation of the task.

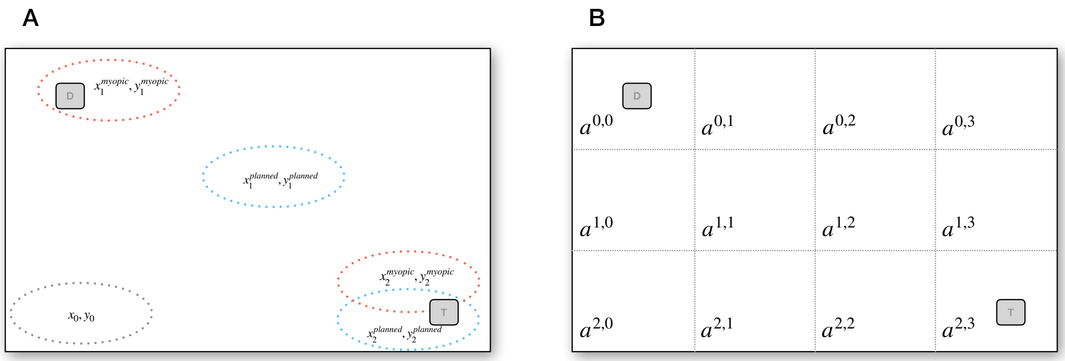


Fig. 2. A. Candidate layout for a visual search task. The user needs to choose between two elements (T and D). If the user is given time to complete two fixations in order to achieve this, there are two strategies she may adopt: 1) first gazing D, followed by T (hypothesised as π_{myopic}), believing D is the most likely target or 2) first gazing in between the targets to collect more information, then moving to the most likely target T (hypothesised $\pi_{planned}$). B. Possible actions in the layout.

3.2 POMDP architecture

The task of the observer is to choose fixation locations (x_i, y_i) to find a target in the visual search environment, taking into account whether they are myopic or planning. We model this task as a POMDP, as this approach has been successful in matching human behaviour in various tasks including visual search (e.g. [3, 6, 7]). In addition, formulation as a POMDP allows accounting for various properties of the task environment (e.g. screen size, number of distractors and item colours), which is required when a graphical layout is in question. We consider two agents defined by their discount rates, a myopic and planning, making decisions in these environments. For the myopic observer, discount rate is near zero, whereas for the planning observer it is near one. As the POMDP, we will consider the the 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R})$, explained below.

States S: The states represent the search task, i.e. the current fixation location and location of the targets T and distractors D.

Actions A: The actions are all fixation locations on the graphical layout. This is simplified by considering the graphical layout as a RxC grid, where each square is one action $a^{R,C}$ (Figure 2B). To reduce the size of the action space, we assume that there are $R * C$ fixation locations (x_i, y_i), which are located in the middle of these squares. The size of each square is the area covered by foveated vision when fixating in the middle of it.

Transitions T: In the current implementation we assume transitions to be deterministic.

Rewards R: Each completed action yields a negative reward, which is the movement time between two locations on the grid. Finding the target yields a larger positive reward.

Observations O: The observer only has partial information about the environment. Hence, observations are initalised as a matrix of the same size as the action space, filled with -1 As the observer gazes to various locations on the layout, they access more information and update this matrix accordingly (0 referring to an empty square, -1 to a distractor and 1 to a target).

We train the POMDP using modified version of the Q-learning Algorithm 2 provided in [7]. This implementation was chosen due to availability of a benchmark source code, and the flexibility of the model to extend it to the various features of the task environment (e.g. item colours, screen size or number of distractors) and agents (e.g. peripheral vision or limitations of the visual system). On the other hand, following an approach similar to presented by Hoppe and Rothkopf [4] using a Belief MDP could be more challenging when extending the model to take into account additional characteristics of the agents, as probabilities of detecting a target would be more ambiguous. In addition, a POMDP approach is a realistic model of actual behaviour. For instance, we can train a model with a target located somewhere on the upper right corner of an environment. Similarly, an actual user may encounter user interface elements, like "Log in" buttons, located in similar locations on various different graphical layouts. Then, when seeing a new interface the agent (or the user) can already guess that trying to locate this item in the familiar location may be a good strategy.

3.3 Myopic and planning observers

As we use Q-learning to learn the values of the states, we distinguish the two observers using the discount rate γ in the POMDP. For the myopic observer, γ is near zero and they only consider short term rewards. On the other hand, the planning observer takes into account future rewards and γ is near one. This contrasts with the approach used in, e.g. [4], where the problem is defined as a belief MDP and policies defined beforehand for the myopic and planning observers.

4 METHODOLOGY

The problem described in section 3 is solved as a POMDP, implemented in Python. This paper focuses on the computational model of visual search, which can be used to identify visual search environments with interesting properties. These layouts can then be used in a user study to validate the computational modelling results, however, this work is outside the scope of this paper.

4.1 Implementation

The computational model used in identifying layouts where the predicted behaviour between a myopic and planning observer differs is based on code presented in [7], implemented in Python 3.8.2. We extend this implementation to the described graphical layouts as the task environment. The full computational model is presented as a Jupyter notebook in appendix. This implementation follows the model presented in section 3.2.

5 RESULTS

This section describes the results of the computational modelling. The exact details of the model are described in the Jupyter notebook provided in appendix. The user study is outside the scope of this paper.

5.1 Simple environments

Environment description. We investigate the computational model in a simple environment, where there is one target in a relatively small screen. Example instances of these types of environments are showed in Figure 3. This environment was investigated under different conditions: namely with 1) different screen sizes, 2) restricted target locations and 3) with static and changing environments in training.

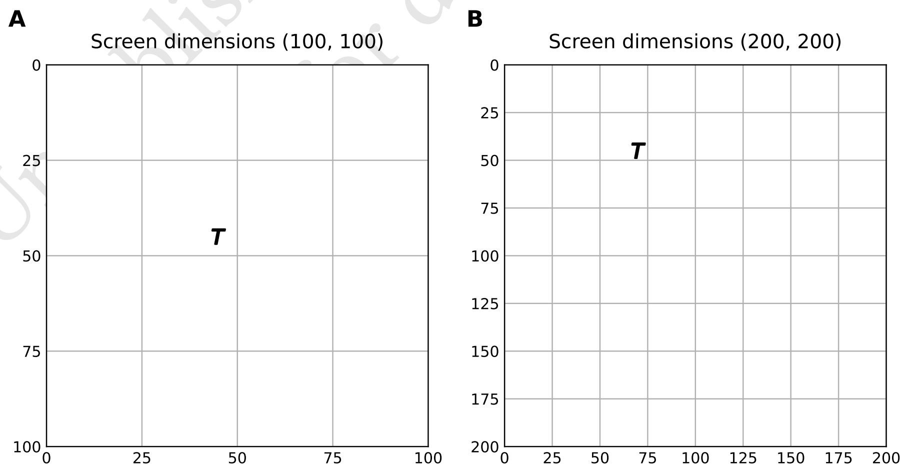


Fig. 3. Two example environments. The grid demonstrates possible fixation locations, i.e. the action space. T is the target. A. Small environment. B. Larger environment. The larger environment has a larger action space.

Myopic and planning observers in the task environment. We observed the behaviour of the myopic and planning observers in simple environments. For the myopic observer, $\gamma = 0.1$ and for the planning observer $\gamma = 0.9$. The results suggest that these environments are too simple for differences to emerge between the two observers. This can most likely be attributed to the task structure, where without taking into account, e.g. peripheral vision, rewards are only allocated at the end of the task. Hence, there are limited intermediary rewards, which would lead to differing behaviour between the two observers. However, we are able to detect several plausible patterns in the data. If the model is trained with the same environment, the observer learns to perform significantly better in this task, in contrast to when the environment is changed in every training episode (Figure 4A). Similarly, the observers perform better on smaller screens (Figure 4C). However, on visual inspection there seems to be no significant difference in cases where the observers are trained on targets, which are restricted to specific areas on the screen (Figure 4B).

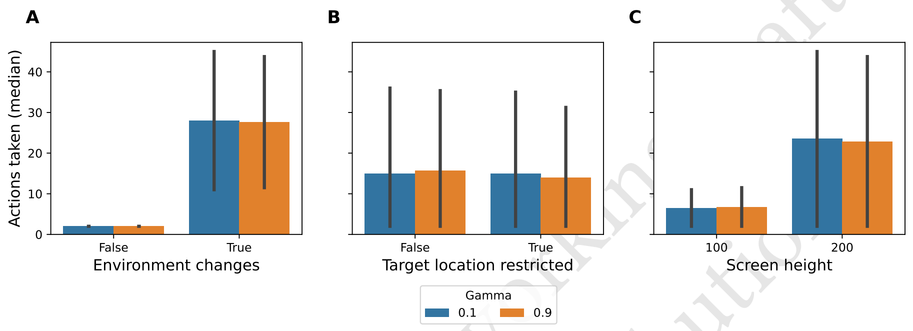


Fig. 4. Prediction results with median number of actions required for the trained observers to complete the visual search task (maximum trials restricted to 100 due to computational reasons). Results displayed for both observers (myopic $\gamma = 0.1$ and planning $\gamma = 0.9$). Results displayed for A. whether environment was changed while training, B. whether target location was restricted to specific area of the screen and C. for different screen sizes.

6 DISCUSSION

This section discusses the results of the computational modelling work. We also propose how these models can be used with real graphical layouts, and which extensions could be beneficial to the existing work.

6.1 Computational modelling results

This paper analysed a planning and a myopic observer in simple visual search environments. Even though we are unable to detect significant differences between a myopic and planning observer in visual search tasks with one target in the environment, without distractors, the model replicates plausible results about real human behaviour. For instance, the learning effect of always performing the same task and the impact of a larger screen size is validated. However, in order to detect differences between the two observers certain additions are required. We propose analysing the same observers in more complex environments, with various distractors, described below.

Myopic and planning observer in complex environments. The computational modelling can be extended to study the observers in more complex environments, including distractors. Examples of such environments are presented in Figure 5. The number of distractors can be modified to correspond to more realistic use cases.

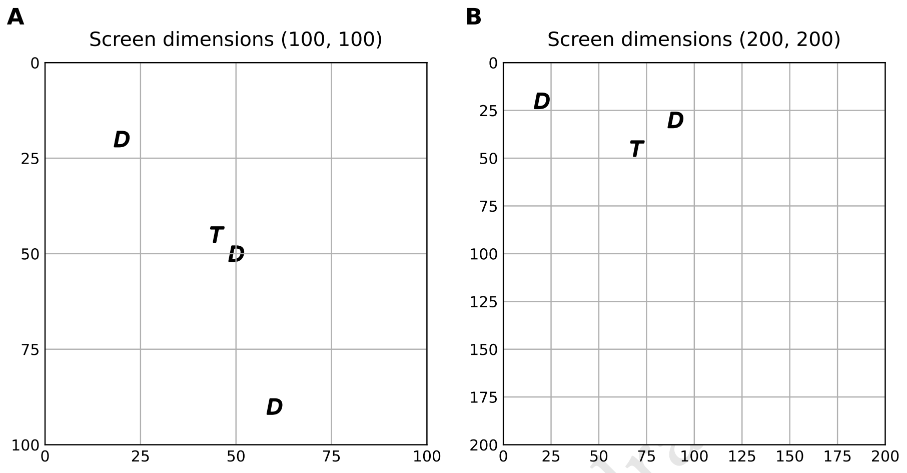


Fig. 5. Two examples of more complex environments. The grid demonstrates possible fixation locations, i.e. the action space. *T* is the target. A. Small environment. B. Larger environment. The screen sizes map to Figure 3.

Analysing the behaviour the two observers in more complex tasks is currently outside the scope of this work. However, it is likely that this type of change is required in order to detect differences between the behaviour of the two observers, as addition of distractors would allow redistribution of rewards. Rewards should be distributed in the middle of the task in addition to the end, so that difference between a myopic and a planning observer can be induced. Finally, considering peripheral vision could lead to detecting layouts where the two observers' behaviour differs.

6.2 Application to real graphical layouts

The proposed computational model can be used in assessing real graphical layouts with a few extensions. Graphical layouts on the web can be framed as environments, where targets are located among distractors. Hence, the simplified version considered in this paper is a sub-category of such realistic graphical layouts. In order to apply the computational model, the elements in the graphical layout need to be classified into targets and distractors (Figure 6A). In the case of real layouts, there may be significantly more distractors than in the simplified version here. Then these layouts, where each element is classified, can be fed into the computational model presented in this paper (Figure 6B). This produces eye-movement strategies that can be expected from a myopic and a planning observer, which can be used in analysing the graphical layout in services such as the AIM server (https://interfacemetrics.aalto.fi/).

6.3 Extensions

The computational modelling approach presented in this paper has certain limitations that could be addressed in future work. Firstly, the graphical layouts currently considered are fairly simple, so future work could incorporate effects of shape, size and colour of different elements on search behaviour. In addition, further constrains of the human visual system could be considered, following approach presented in [3].

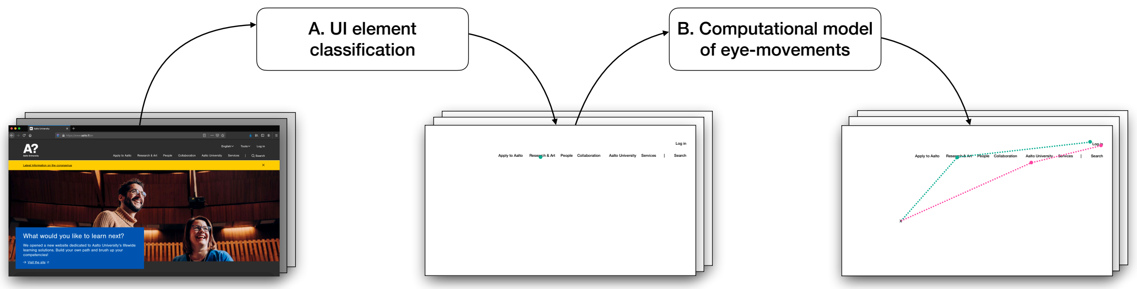

A. UI element classification
B. Computational model of eye-movements

Fig. 6. Application of the proposed computational model in real graphical layouts. Elements of a graphical layout are classified into targets and distractors (A). Then these classified layouts are fed into the computational model presented in this paper (B) to produce eye-movement strategies.

7 CONCLUSION

We compared two strategies, myopic and planning, in visual search tasks in graphical layouts. The strategies were defined based on the discount factor γ in the POMDP formulation, where values near zero indicate a myopic and values near one a planning observer. We presented a computational model for generating graphical layouts, where the strategies of these observers can be investigated. The computational modelling results suggest that further analysis is required to detect significantly different strategies for the observers emerging as a result of the environment. This may be achieved by considering further features of a user interface in the graphical layouts (e.g. item colours) or the observers (e.g. constraints of the visual system). However, the current computational model can be used as a baseline for analysing potential eye-movement strategies in real graphical layouts.

8 APPENDICES

The notebook used in computational modelling is attached to the end of this document.

REFERENCES

[1] A. Acharya, X. Chen, Christopher W. Myers, R. L. Lewis, and Andrew Howes. 2017. Human Visual Search as a Deep Reinforcement Learning Solution to a POMDP. (2017).

[2] N. J. Butko and J. R. Movellan. 2008. I-POMDP: An infomax model of eye movement. In *2008 7th IEEE International Conference on Development and Learning*. 139–144. https://doi.org/10.1109/DEVLRN.2008.4640819

[3] Xiuli Chen, Aditya Acharya, Antti Oulasvirta, and Andrew Howes. 2021. An Adaptive Model of Gaze-Based Selection. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 288, 11 pages. https://doi.org/10.1145/3411764.3445177

[4] David Hoppe and Constantin A. Rothkopf. 2019. Multi-step planning of eye movements in visual search. *Scientific Reports* 9, 1 (Dec. 2019), 144. https://doi.org/10.1038/s41598-018-37536-0

[5] Laurent Itti and Christof Koch. 2000. A saliency-based search mechanism for overt and covert shifts of visual attention. *Vision Research* 40, 10 (2000), 1489 – 1506. https://doi.org/10.1016/S0042-6989(99)00163-7

[6] Jussi P.P. Jokinen, Zhenxin Wang, Sayan Sarcar, Antti Oulasvirta, and Xiangshi Ren. 2020. Adaptive feature guidance: Modelling visual search with graphical layouts. *International Journal of Human-Computer Studies* 136 (2020), 102376. https://doi.org/10.1016/j.ijhcs.2019.102376

[7] Jussi P. P. Jokinen, Tuomo Kujala, and Antti Oulasvirta. 0. Multitasking in Driving as Optimal Adaptation Under Uncertainty. *Human Factors* 0, 0 (0), 0018720820927687. https://doi.org/10.1177/0018720820927687 arXiv:https://doi.org/10.1177/0018720820927687 PMID: 32731763.

[8] Jiri Najemnik and Wilson S. Geisler. 2005. Optimal eye movement strategies in visual search. *Nature* 434, 7031 (March 2005), 387–391. https://doi.org/10.1038/nature03390

[9] Jiri Najemnik and Wilson S. Geisler. 2009. Simple summation rule for optimal fixation selection in visual search. *Vision Research* 49, 10 (2009), 1286–1294. https://doi.org/10.1016/j.visres.2008.12.005 Visual Attention: Psychophysics, electrophysiology and neuroimaging.

Sequential Planning in Visual Search Tasks with Graphical Layouts

Aini Putkonen - ELEC-E7861 Research Project in Human Computer Interaction

This notebook presents two Partially Observable Markov Decision Process (POMDP) models (myopic and planning) used in evaluating graphical layouts for a visual search task. The idea of contrasting myopic and planning policies is based on a belief MDP implementation by Hoppe & Rothkopf (2019) (https://www.nature.com/articles/s41598-018-37536-0). The implementation as POMDP is based on source code presented in Jokinen et al. (2020) (https://doi.org/10.1177%2F0018720820927687), available here (https://gitlab.com/jokinenj/multitasking-driving/-/tree/master/obs-prob).

Table of contents

Problem description and POMDP formulation

Problem description

The purpose of the presented computational models is to identify layouts where sequential planning may be useful. i.e. we contrast two observers: one only taking account short term rewards (myopic) and another considering long term rewards (planning).

An example of a visual search task on a user interface is detecting target elements T, such as buttons and input fields, in the midst of distractors D. This is illustrated in Figure 1. In some cases, a user may find beneficial gazing directly to an element they believe to be the most probable target location ($myopic$ in Figure 1). On the other hand, in other layouts gazing between elements first, followed by a saccade to the most likely target may be a better strategy ($planned$ in Figure 1).

POMDP formulation

This type of behaviour in graphical layouts can be represented as a POMDP, which is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \mathcal{Z})$. The formulation is explained below.

States S***:*** The states represent the search task, i.e. the current fixation location and location of the targets T and distractors D

Actions A***:*** The actions are all fixation locations on the graphical layout. This is simplified by considering the graphical layout as a RxC grid, where each square is one action $a^{R,C}$. To reduce the size of the action space, we assume that there are $R * C$ fixation locations (x_i, y_i), which are located in the middle of these squares. The size of each square is the area covered by foveated vision when fixating in the middle of it.

Transitions T***:*** In the current implementation we assume transitions to be deterministic.

Rewards R***:*** Each completed action yields a negative reward, which is the movement time between two locations on the grid. Finding the target yields a larger positive reward.

Observations O***:*** The observer only has partial information about the environment. Hence, observations are initalised as a matrix of the same size as the action space, filled with -1 As the observer gazes to various locations on the layout, they access more information and update this matrix accordingly (0 referring to an empty square, -1 to a distractor and 1 to a target).

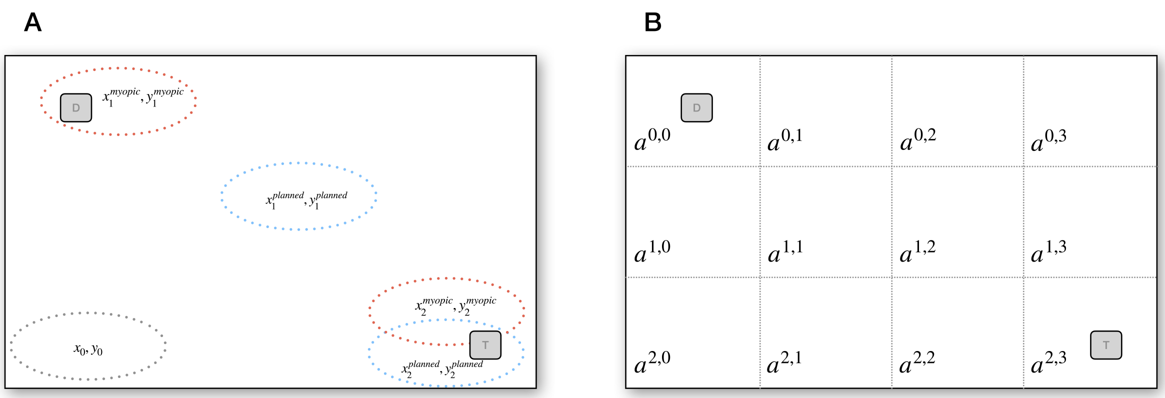


Figure 1: A. Two strategies for choosing two fixation locations (x_i, y_i) in detecting a target T amongst distractors D on a layout. A myopic observer first gazes the most probable target location. A planned observer may collect more information by first gazing between targets. B. The action space represented as an RxC grid, where each square represents one possible action $a^{R,C}$.

Import libraries

In [1]:
```python
from math import ceil, floor, sqrt
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import matplotlib.patches as patches
import tqdm
from operator import itemgetter
import math
from pprint import pprint
import pandas as pd
import seaborn as sns
import string

%config InlineBackend.figure_format = 'retina'

np.random.seed(123)
```

Define the external environment

Here we define the external environment for the visual search task, i.e. the graphical layout where the search takes place.

Action matrix

The number of potential actions is set based on the size of the screen of the layout and area covered by one fixation. We assume that the graphical layout can be divided into a grid, where each square represents a potential fixation location. Thus, we define the action space as a matrix, where fixating on each square on the grid is one action with a unique index. E.g. for a layout with six potential actions:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

Task matrix

The details of this environment (i.e. the location of targets and possible distractors) are unavailable to the agent at the time of performing the task. We define the task as a matrix, where each cell can take the following values:

Value	Description
-1	Distractor (optional)
0	Empty
1	Target

Hence, if the target is at action location 2, the task matrix takes the form:

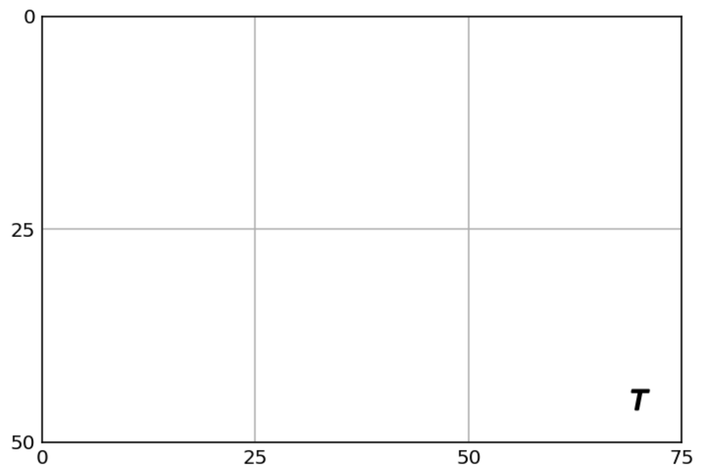
$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

In [2]:
```
class VisualSearchEnv:

    '''
    A class used to represent the external visual search environment


    Attributes
    ----------
    screen_height : float
        height of the screen where graphical layout is displayed
    screen_width : float
        width of the screen where graphical layout is displayed
    target_loc : tuple
        (x,y) coordinates of the target location (in px)
    distractor_loc : tuple
        (x, y) coordinates of the distractor location

    Methods
    -------
    set_action_matrix
        create the action matrix
    set_task
        create the task matrix including target (and distractor) locations
    init_lookup_table
        initialise lookup table for distances between two actions (locations
    draw_env(action_space=False)
        plot a visual representation of the external environment (graphical l
    '''

    def __init__(self, screen_height: float, screen_width: float, target_loc:

        self.screen_height = screen_height
        self.screen_width = screen_width
        self.target_loc = target_loc
        self.distractor_loc = distractor_loc

        self.set_action_matrix()
        self.set_task()

    def set_action_matrix(self):

        self.fixation_height = 25 # Height of fixation in px
        self.fixation_width = 25 # Width of fixation in px

        try:
            self.n_rows = ceil(self.screen_height / self.fixation_height)
            self.n_cols = ceil(self.screen_width / self.fixation_width)

            if self.n_rows == 0 or self.n_cols == 0:
                raise Exception('Too small screen!')

        except Exception as e:
            print (str(e))

        num = 0
        action_matrix = np.zeros((self.n_rows, self.n_cols), dtype=int)
        for row in range(self.n_rows):
            for col in range(self.n_cols):
                action_matrix[row,col] = num
                num += 1

        self.action_matrix = action_matrix # Location of targets (and distrac

    def set_task(self):

        self.task = np.zeros((self.action_matrix.shape), dtype = int)

        tgt_x_on_grid = floor(self.target_loc[0] / self.fixation_height)
        tgt_y_on_grid = floor(self.target_loc[1] / self.fixation_width)

        if self.distractor_loc:
            for distractor in self.distractor_loc:
                dtr_x_on_grid = floor(distractor[0] / self.fixation_height)
                dtr_y_on_grid = floor(distractor[1] / self.fixation_width)
                self.task[dtr_y_on_grid, dtr_x_on_grid] = -1

        self.task[tgt_y_on_grid, tgt_x_on_grid] = 1

    def init_lkp_table(self):

        actions = self.action_matrix.flatten()
        n_actions = len(actions)
        self.lkp_table = np.zeros((n_actions, n_actions))

        for previous_action in actions:
            previous_action_idx = np.where(self.action_matrix == previous_act

            for next_action in actions:
                next_action_idx = np.where(self.action_matrix == next_action)

                v_dist = abs(previous_action_idx[0]-next_action_idx[0])*self.
                h_dist = abs(previous_action_idx[1]-next_action_idx[1])*self.

                self.lkp_table[previous_action, next_action] = np.sqrt(v_dist

        return self.lkp_table

    def draw_env(self, ax: matplotlib.axes.Axes = None, action_space: bool = 

        """
        Plot a visual representation of the external environment (graphical l

        Parameters
        ----------
        action_space : bool
            Boolean indicating whether action space grid will be shown on the
        """

        if ax == None:
            fig, ax = plt.subplots(1)

        ax.set_aspect(aspect='equal')
        ax.tick_params(which='major', length=0)

        tgt_x = self.target_loc[0]
        tgt_y = self.target_loc[1]
        ax.scatter(tgt_x, tgt_y, marker = '$T$', s = 100, c='black') # Target

        if self.distractor_loc != None:
            for distractor in self.distractor_loc:
                dtr_x = distractor[0]
                dtr_y = distractor[1]
                ax.scatter(dtr_x, dtr_y, marker = '$D$', s = 100, c='black')

        if action_space:

            grid_width = self.fixation_width

            grid_height = self.fixation_height

            ax.xaxis.set_ticks([x*grid_width for x in range(0,self.n_cols+1)]
            ax.yaxis.set_ticks([x*grid_height for x in range(0,self.n_rows+1)
            ax.grid(True)

        ax.set_xlim(0, self.screen_width)
        ax.set_ylim(self.screen_height, 0)

        return ax
```

An example of a visual search environment instance is presented below.

In [3]:
```
demo_env = VisualSearchEnv(screen_height = 50, screen_width = 75, \
                           target_loc = (70, 45),)

demo_env.draw_env(action_space=True)
```

Out[3]:
```
<AxesSubplot:>
```



We also want to generate a lookup table for the distances between two cells, to be used later in calculating movement times. Each row and column is one action.

In [4]:
```
demo_env.init_lkp_table()
print (demo_env.lkp_table)
```

```
[[ 0.         25.         50.         25.         35.35533906 55.90169944]
 [25.          0.         25.         35.35533906 25.         35.35533906]
 [50.         25.          0.         55.90169944 35.35533906 25.        ]
 [25.         35.35533906 55.90169944  0.         25.         50.        ]
 [35.35533906 25.         35.35533906 25.          0.         25.        ]
 [55.90169944 35.35533906 25.         50.         25.          0.        ]]
```

Define observations

The properties of the external environment are not directly observable by the agent, and they have to collect information about it as they move their gaze in the environment. Initially the agent has no information about the environment, i.e. they do not know what each location in the external environment contains. We generate an observation matrix, which takes the same shape as the action matrix. The observation matrix can take the following values:

Value	Description
-1	No information
0	No target
1	Target

The observation matrix is initialised with values of -1, and at each iteration the cells in the observation matrix are updated to correct values. Thus, at the beginning of training the observation matrix takes the form:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

If the observer takes the action 1 and there's nothing in this cell, the observation table updates to:

$$\begin{bmatrix} -1 & 0 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

In [5]:
```
class Observation:

    '''
    A class used to represent the observation space

    Methods
    -------
    initialise_obs
        initialise observation matrix, with fill value -1
    update_obs_matrix
        update observation matrix as more information is collected about the
    '''

    def __init__(self, shape: tuple):
        self.shape = shape
        self.initialise_obs()

    def initialise_obs(self):
        self.obs_matrix = np.full(shape=self.shape, fill_value=-1)

    def update_obs_matrix(self, loc: tuple, is_target: bool):
        self.obs_matrix[loc] = 1 if is_target else 0
```

Define transitions

In this implementation we assume transitions to be deterministic. However, there may be some inaccuracies in the transitions when humans perform a saccade.

In [6]:
```
# Assuming deterministic transitions for now
```

Display a candidate environment and corresponding actions, task and observations

Here we print an example of the action, task and observation matrices, with their corresponding environment, for visual inspection.

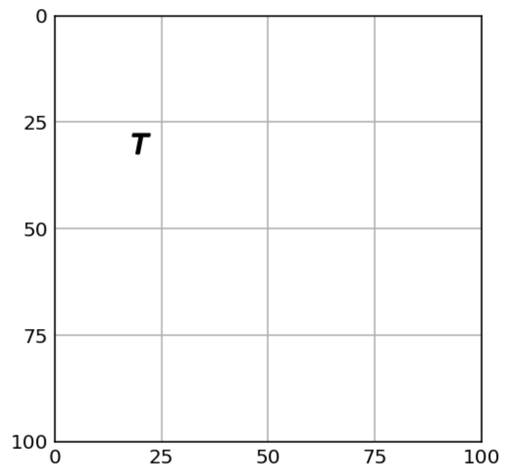
In [7]:
```
SCREEN_HEIGHT = 100 # Screen height in px
SCREEN_WIDTH = 100 # Screen width in px

try:
    TARGET_LOC = (20,30)

    if TARGET_LOC[0] > SCREEN_WIDTH or TARGET_LOC[1] > SCREEN_HEIGHT:
        raise Exception("Target outside of screen!")
except Exception as e:
    print (str(e))

try:
    #DISTRACTOR_LOC = [(20, 20), (30, 50)]
    DISTRACTOR_LOC = []

    for distractor in DISTRACTOR_LOC:
        if distractor[0] > SCREEN_WIDTH or distractor[1] > SCREEN_HEIGHT:
                raise Exception("Distractor outside of screen!")

except Exception as e:
    print (str(e))

env = VisualSearchEnv(screen_height = SCREEN_HEIGHT, screen_width = SCREEN_WI
                      target_loc = TARGET_LOC, distractor_loc = DISTRACTOR_LC

env.set_action_matrix()
env.set_task()

print ('---------------------')
print ('ACTIONS')
print ('---------------------')
print (env.action_matrix)

print ('---------------------')
print ('TASK')
print ('---------------------')
print (env.task)

obs = Observation(env.action_matrix.shape)
obs.initialise_obs()

print ('---------------------')
print ('INITIAL OBSERVATIONS')
print ('---------------------')
print (obs.obs_matrix)
```

```
--------------------
ACTIONS
--------------------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
--------------------
TASK
--------------------
[[0 0 0 0]
```

```
 [1 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
--------------------
INITIAL OBSERVATIONS
--------------------
[[-1 -1 -1 -1]
 [-1 -1 -1 -1]
 [-1 -1 -1 -1]
 [-1 -1 -1 -1]]
```

In [8]:
```
env.draw_env(action_space = True)
```

Out[8]:
```
<AxesSubplot:>
```



Set Up Q-learning

The algorithm used in Q-learning is defined in the cell below. This algorithm is based on [Jokinen et al. (2020) (https://doi.org/10.1177%2F0018720820927687)](https://doi.org/10.1177%2F0018720820927687), source code available [here (https://gitlab.com/jokinenj/multitasking-driving/-/tree/master/obs-prob)](https://gitlab.com/jokinenj/multitasking-driving/-/tree/master/obs-prob), modified to the environment described above.

NOTE: Currently the movement time calculations are only indicative, as the angles required in EMMA are not matched to realistic graphical layouts.

In [9]:
```
class QLearning:

    '''
    A class used to run Q-learning for a given external environment.
    Based on https://gitlab.com/jokinenj/multitasking-driving/-/blob/master/c

    Attributes
    ----------
    gamma: float
        discount rate determining whether the observer is myopic or planning
    screen_height: int
        screen height of the environment where Q-learning is run (we train wi
    screen_width: int
        screen width of the environment where Q-learning is run (we train wit
    peripheral_vision: bool
        indicate whether the consider peripheral vision in training
    change_env: bool
        indicate whether to change env in training

    Methods
    -------
    reset
        reset Q-learning at each episode
    get_action_loc
        get (row, col) of an action index in the action matrix
    observe
        update observation table based on feedback from task environment
    set_belief_state
        set a new belief state based on gaze location and observation + initi
    calculate_max_q_value
    choose_action_softmax
        choose action using a softmax policy
    weighted_random
    update_q_sarsa
        update Q-table
    EMMA_fixation_time
        calculate movement time for fixation
    calculate_mt
        compute movement time from previous location to current
    calculate_reward
        compute reward for a given observation
    choose_action_random
        choose action using a random policy
    iterate
        perform one iteration of Q-learning
    '''

    def __init__(self, gamma, screen_height, screen_width, restricted_tgt = F
                 peripheral_vision=False, change_env=False):

        self.change_env = change_env
        self.envs = []

        self.peripheral_vision = peripheral_vision
        self.restricted_tgt = restricted_tgt

        self.q = {}
        self.gamma = gamma

        self.softmax_temp = 0.5 #Softmax temperature
        self.alpha = 1
        self.eps = 0.1
        self.learning = True
        self.cell_distance = 5

        self.found_reward = 2000

        self.screen_height = screen_height
        self.screen_width = screen_width

        self.reset(init=True)

    def reset(self, init=False):

        if self.change_env or init:
            self.env, self.observation = self.create_env()
            self.envs.append(self.env)

        if init:
            self.lkp_table = self.env.init_lkp_table()

        self.actions = self.env.action_matrix
        self.task = self.env.task

        self.obs = self.observation.obs_matrix
        self.done = False # Determines whether task is done
        self.n_iter = 0 # Keeps count of time horizon

        self.choose_action_random()
        #self.action = 0
        self.eye_loc = self.action

        self.reward = 0
        self.mt = 0

        self.set_belief_state()
        self.reward = 0

    def create_env(self):

        SCREEN_HEIGHT = self.screen_height
        SCREEN_WIDTH = self.screen_width

        if self.restricted_tgt:
            TARGET_LOC = (np.random.uniform(SCREEN_WIDTH - SCREEN_WIDTH/10, S
                          np.random.uniform(SCREEN_HEIGHT - SCREEN_WIDTH/10,
        else:
            TARGET_LOC = (np.random.uniform(0, SCREEN_WIDTH), np.random.unifo

        env = VisualSearchEnv(screen_height = SCREEN_HEIGHT, screen_width = S
                      target_loc = TARGET_LOC,) #distractor_loc = DISTRACTOR_


        obs = Observation(env.action_matrix.shape)
        return env, obs

    def get_action_loc(self):
        return np.where(self.actions == self.action)

    def observe(self):

        # Find the coordinates of the action
        action_loc = self.get_action_loc()

        if self.peripheral_vision:
            start_row = action_loc[0] - 1
            start_col = action_loc[1] + 1

            for row in range(start_row[0], start_row[0] + 3):
                for col in range(start_col[0], start_col[0] + 3):
                    try:
                        self.obs[row, col] = self.task[row, col]
                    except:
                        continue

        else:
            self.obs[action_loc] = self.task[action_loc]

        return self.obs

    def set_belief_state(self):
        # Must turn belief state into string, otherwise too complicated struc
        self.belief = repr([self.eye_loc,self.obs])
        # Update Q
        if self.belief not in self.q:
            self.q[self.belief] = {}
            for action in self.actions.flatten():
                self.q[self.belief][action] = 0.0

    def calculate_max_q_value(self):
        return max(self.q[self.belief].items(), key=itemgetter(1))

    def choose_action_softmax(self):

        if self.softmax_temp == 0:
            self.action = self.calculate_max_q_value()[0]
            return self.action

        p = {}

        try:
            for a in self.q[self.belief].keys(): # Iterate over actions
                p[a] = math.exp(self.q[self.belief][a] / self.softmax_temp)
        except OverflowError:
            self.action = self.calculate_max_q_value()[0]
            return self.action

        s = sum(p.values())

        if s != 0:
            p = {k: v/s for k,v in p.items()}
```

```
                self.action = self.weighted_random(p)
            else:
                self.action = np.random.choice(list(p.keys()))

    def weighted_random(self, weights):
        number = np.random.random() * sum(weights.values())
        for k,v in weights.items():
            if number < v:
                break
            number -= v
        return k

    def update_q_sarsa(self):
        if self.learning:
            previous_q = self.q[self.previous_belief][self.previous_action]
            next_q = self.q[self.belief][self.action]
            self.q[self.previous_belief][self.previous_action] = \
                previous_q + self.alpha * (self.reward + self.gamma * next_q

    # Eye movement and encoding time come from EMMA (Salvucci, 2001).
    def EMMA_fixation_time(self, distance, freq = 0.1, encoding_noise = False
        emma_KK = 0.006
        emma_k = 0.4
        emma_prep = 0.135
        emma_exec = 0.07
        emma_saccade = 0.002
        E = emma_KK * -math.log(freq) * math.exp(emma_k * distance)
        if encoding_noise:
            E += np.random.gamma(E, E/3)
        if E < emma_prep: return E
        S = emma_prep + emma_exec + emma_saccade * distance
        if (E <= S): return S
        E_new = (emma_k * -math.log(freq))
        if encoding_noise:
            E_new += np.random.gamma(E_new, E_new/3)
        T = (1 - (S / E)) * E_new
        return S + T

    def calculate_reward(self):
        action_loc = self.get_action_loc()
        # Target found.
        if self.task[action_loc] == 1:
            self.mt += 0.15 # add motor movement time for response
            self.reward = self.found_reward - self.mt
            self.done = True
        else:
            # Target not yet found.
            self.reward = -self.mt

        if self.peripheral_vision:
            start_row = action_loc[0] - 1
            start_col = action_loc[1] + 1

            for row in range(start_row[0], start_row[0] + 3):
                for col in range(start_col[0], start_col[0] + 3):
                    try:
                        if self.task[row, col] == 1:
                            self.reward += 10
                        #self.done = True
                except:
                    continue


    def choose_action_random(self):
        self.action = np.random.choice(self.actions.flatten())

    def iterate(self):

        self.previous_belief = self.belief
        self.observe()
        self.set_belief_state()

        self.previous_action = self.action
        self.choose_action_softmax()
        self.eye_loc = self.action

        self.update_q_sarsa()

        eccentricity = self.lkp_table[self.previous_action, self.action] # No
        self.mt = self.EMMA_fixation_time(eccentricity*self.cell_distance, en

        self.calculate_reward()


    def predict(self, env):

        self.env = env
        self.reset()

        max_tries = 100

        actions_saved = []


        count = 0
        while True:

            self.choose_action_softmax()
            #if np.random.random() < self.eps:
            #    action = max(self.q[self.belief].items(), key=itemgetter(1))
            #else:
            #    action = np.random.choice(self.actions.flatten())

            actions_saved.append(self.action)

            if np.where(self.actions == self.action) == np.where(self.task ==
                return actions_saved, True

            if count > max_tries:
                return actions_saved, False

            count += 1
```

Iterate and predict

We then iterate over the Q-learning algorithm for the myopic and planning observers.

In [10]:
```
# Define hyperparameters for training and prediction
n_episodes = 1000
n_predictions = 1000
```

Here we define the different conditions for which training is run. We consider:

- Two gammas (myopic=0.1, planning=0.9) in `gammas`
- Different screen dimensions `screen_dims`
- Whether target is always restricted to some location on the user interface `restricted_tgts`
- Whether training is run with changing environments (where target changes place), or with one `change_envs`

In [11]:
```
gammas = [0.1, 0.9]
screen_dims = [100, 200]
restricted_tgts = [True, False]
change_envs = [True, False]
```

In [12]:
```
q_learning_instances = []

for gm in gammas:
    for screen_dim in screen_dims:
        for restricted_tgt in restricted_tgts:
            for env in change_envs:
                q_learning = QLearning(gamma = gm, screen_height = screen_dim
                                       restricted_tgt=restricted_tgt, change_
                                       peripheral_vision = False)
                obs.initialise_obs()
                for episode in tqdm.trange(n_episodes):
                    q_learning.reset()
                    while not q_learning.done:
                        q_learning.iterate()

                q_learning_instances.append(q_learning)

#pprint(q_learning.q)
```

```
100%|██████████| 1000/1000 [00:01<00:00, 683.61it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2056.66it/s]
100%|██████████| 1000/1000 [00:02<00:00, 372.31it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1964.18it/s]
100%|██████████| 1000/1000 [00:19<00:00, 52.18it/s]
100%|██████████| 1000/1000 [00:02<00:00, 337.08it/s]
100%|██████████| 1000/1000 [00:19<00:00, 50.45it/s]
100%|██████████| 1000/1000 [00:02<00:00, 416.05it/s]
100%|██████████| 1000/1000 [00:01<00:00, 635.69it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2131.60it/s]
100%|██████████| 1000/1000 [00:02<00:00, 356.24it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1843.89it/s]
100%|██████████| 1000/1000 [00:19<00:00, 51.33it/s]
100%|██████████| 1000/1000 [00:02<00:00, 445.57it/s]
100%|██████████| 1000/1000 [00:18<00:00, 54.29it/s]
100%|██████████| 1000/1000 [00:02<00:00, 455.57it/s]
```

In [13]:
```
# Define what we want to save from predictions
actions = []
action_lens_median = []
#success = []
gammas = []
scrn_heights = []
scrn_widths = []
restricted_tgts = []
change_envs = []
```

In [14]:
```python
for instance in q_learning_instances:
    action_lens = np.zeros(n_predictions)

    for prediction in tqdm.trange(n_predictions):
        instance.softmax = 0.1
        actions_saved, success = instance.predict(instance.env)
        action_lens[prediction] = len(actions_saved)

    action_lens_median.append(np.median(action_lens))
    gammas.append(instance.gamma)
    scrn_heights.append(instance.screen_height)
    scrn_widths.append(instance.screen_width)
    restricted_tgts.append(instance.restricted_tgt)
    change_envs.append(instance.change_env)

q_learning_data = pd.DataFrame({'action_lens_median': action_lens_median,
                     'gamma': gammas,
                     'scrn_height': scrn_heights,
                     'scrn_width': scrn_widths,
                     'restricted_tgt': restricted_tgts,
                     'change_env': change_envs})
```

```
100%|██████████| 1000/1000 [00:00<00:00, 1851.41it/s]
100%|██████████| 1000/1000 [00:00<00:00, 4900.25it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1975.60it/s]
100%|██████████| 1000/1000 [00:00<00:00, 5961.39it/s]
100%|██████████| 1000/1000 [00:02<00:00, 394.17it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2495.41it/s]
100%|██████████| 1000/1000 [00:02<00:00, 405.96it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2282.43it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2235.87it/s]
100%|██████████| 1000/1000 [00:00<00:00, 5779.91it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2004.82it/s]
100%|██████████| 1000/1000 [00:00<00:00, 6516.61it/s]
100%|██████████| 1000/1000 [00:02<00:00, 464.44it/s]
100%|██████████| 1000/1000 [00:00<00:00, 3013.16it/s]
100%|██████████| 1000/1000 [00:02<00:00, 449.24it/s]
100%|██████████| 1000/1000 [00:00<00:00, 3058.22it/s]
```

Analyse and plot data

The results from predictions are displayed below.

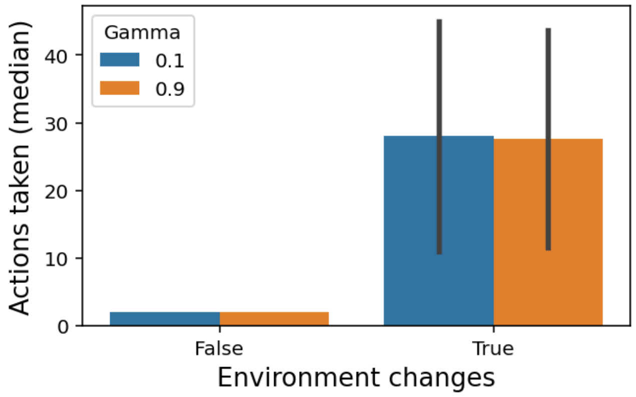
In [15]:
```
display(q_learning_data)
```

	action_lens_median	gamma	scrn_height	scrn_width	restricted_tgt	change_env
0	10.0	0.1	100	100	True	True
1	2.0	0.1	100	100	True	False
2	12.0	0.1	100	100	False	True
3	2.0	0.1	100	100	False	False
4	46.0	0.1	200	200	True	True
5	2.0	0.1	200	200	True	False
6	44.0	0.1	200	200	False	True
7	2.0	0.1	200	200	False	False
8	11.0	0.9	100	100	True	True
9	2.0	0.9	100	100	True	False
10	12.0	0.9	100	100	False	True
11	2.0	0.9	100	100	False	False
12	41.0	0.9	200	200	True	True
13	2.0	0.9	200	200	True	False
14	46.5	0.9	200	200	False	True
15	2.0	0.9	200	200	False	False

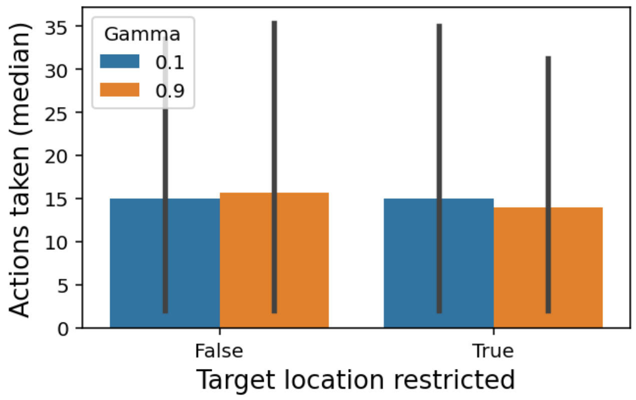
The results are plotted below.

In [16]:
```
# Define some parameters for plotting
label_size = 13
```

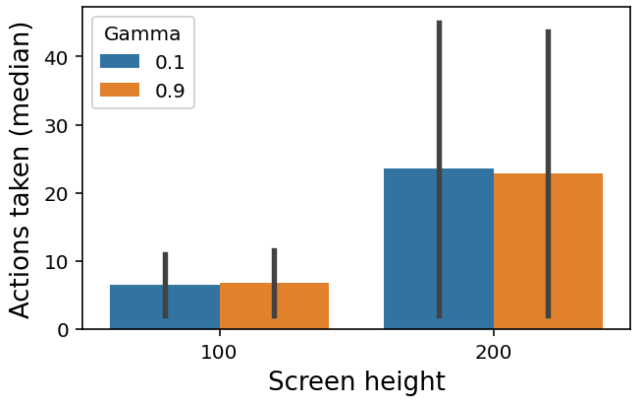
In [17]:
```python
def plot_change_env(ax,legends=True):
    sns.barplot(x="change_env", y="action_lens_median", hue="gamma", data=q_l

    ax.set_xlabel("Environment changes", size = label_size)
    ax.set_ylabel("Actions taken (median)", size = label_size)
    ax.legend(loc='upper left', title = 'Gamma')

    if not legends:
        ax.legend().set_visible(False)

fig, ax1 = plt.subplots(figsize = (5,3))
plot_change_env(ax1)
```



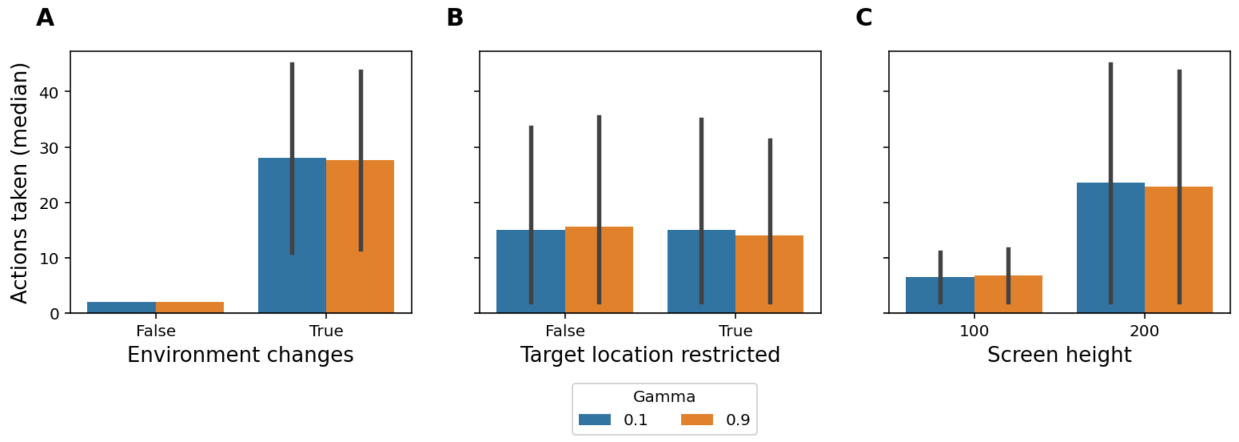
In [18]:
```python
def plot_restricted_tgt(ax, legends=True):

    sns.barplot(x="restricted_tgt", y="action_lens_median", hue="gamma", data

    ax.set_xlabel("Target location restricted", size = label_size)
    ax.set_ylabel("Actions taken (median)", size = label_size)
    ax.legend(loc='upper left', title = 'Gamma')

    if not legends:
        ax.legend().set_visible(False)

fig, ax2 = plt.subplots(figsize = (5,3))
plot_restricted_tgt(ax2)
```



In [19]:
```
def plot_screen_height(ax, legends=True):

    sns.barplot(x="scrn_height", y="action_lens_median", hue="gamma", data=q_

    ax.set_xlabel("Screen height", size = label_size)
    ax.set_ylabel("Actions taken (median)", size = label_size)
    ax.legend(loc='upper left', title = 'Gamma')

    if not legends:
        ax.legend().set_visible(False)

fig, ax3 = plt.subplots(figsize = (5,3))
plot_screen_height(ax3)
```

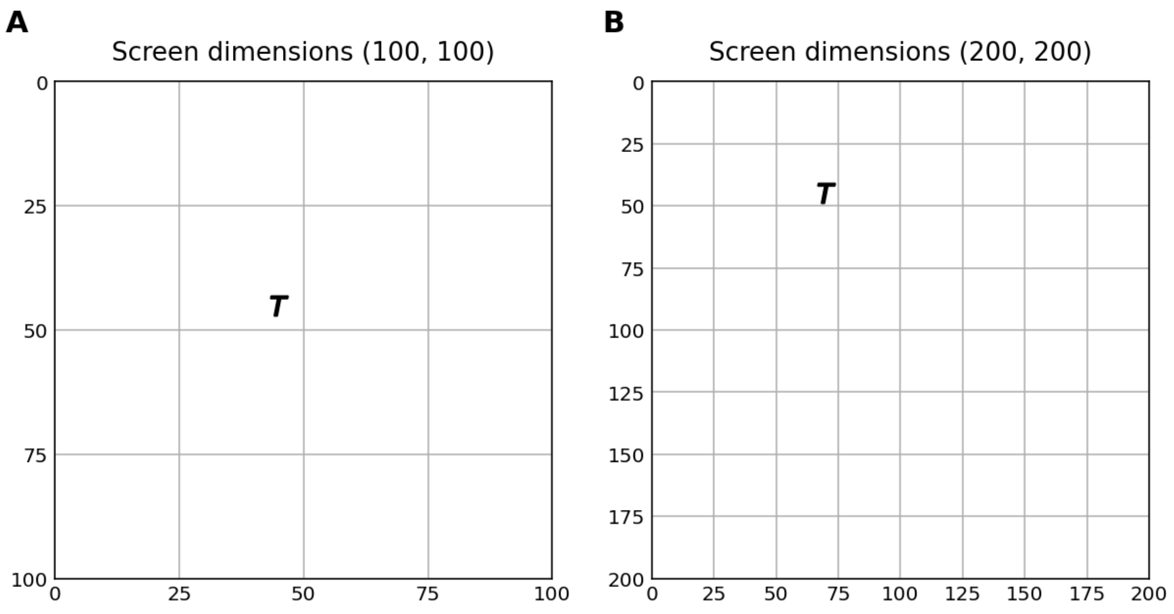


In [20]:
```
fig, axs = plt.subplots(1,3, figsize = (13,3), sharey=True)

for i in range(3):
    axs[i].text(-0.1, 1.1, string.ascii_uppercase[i], transform=axs[i].transA

plot_change_env(axs[0], legends=False)
plot_restricted_tgt(axs[1], legends=False)
plot_screen_height(axs[2], legends=False)

axs[1].yaxis.label.set_visible(False)
axs[2].yaxis.label.set_visible(False)

handles, labels = axs[0].get_legend_handles_labels()
fig.legend(handles, labels, loc='lower left', title = 'Gamma', ncol=2, bbox_t
           fontsize = 'medium')

plt.savefig("figs/results_simple_env.pdf", dpi=300, bbox_inches = 'tight')
```



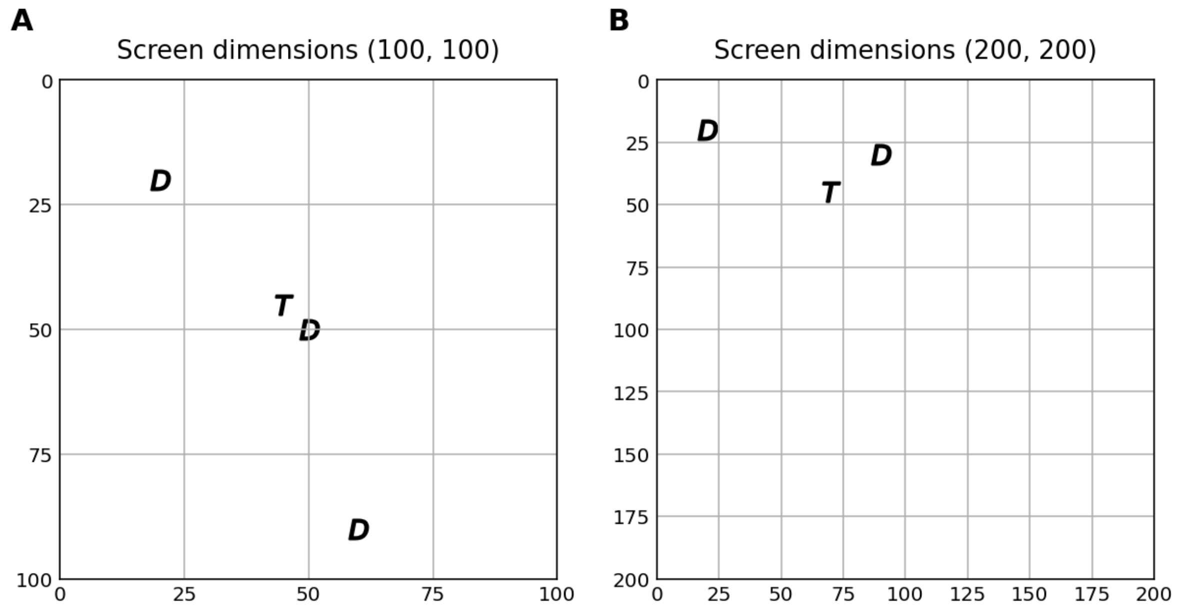
Finally, plot some examples of environments. First, a simple environment.

In [21]:
```
fig, axs = plt.subplots(1,2, figsize = (10,10))

for i in range(2):
    axs[i].text(-0.1, 1.1, string.ascii_uppercase[i], transform=axs[i].transA

small = 100
demo_env1 = VisualSearchEnv(screen_height = small, screen_width = small, \
                      target_loc = (45, 45),)

big = 200
demo_env2 = VisualSearchEnv(screen_height = big, screen_width = big, \
                      target_loc = (70, 45),)

demo_env1.draw_env(ax = axs[0], action_space=True)
axs[0].set_title(f'Screen dimensions ({small}, {small})', size = label_size,

demo_env2.draw_env(ax = axs[1], action_space=True)
axs[1].set_title(f'Screen dimensions ({big}, {big})', size = label_size, y= 1

plt.savefig("figs/env_example.pdf", dpi=300, bbox_inches = 'tight')
```



Then a more complex environment for reference.

In [22]:
```
fig, axs = plt.subplots(1,2, figsize = (10,10))

for i in range(2):
    axs[i].text(-0.1, 1.1, string.ascii_uppercase[i], transform=axs[i].transA

small = 100
demo_env1 = VisualSearchEnv(screen_height = small, screen_width = small, \
                            target_loc = (45, 45), distractor_loc = [(20, 20), (50,

big = 200
demo_env2 = VisualSearchEnv(screen_height = big, screen_width = big, \
                            target_loc = (70, 45), distractor_loc = [(20, 20), (90,

demo_env1.draw_env(ax = axs[0], action_space=True)
axs[0].set_title(f'Screen dimensions ({small}, {small})', size = label_size,

demo_env2.draw_env(ax = axs[1], action_space=True)
axs[1].set_title(f'Screen dimensions ({big}, {big})', size = label_size, y= 1

plt.savefig("figs/env_example_complex.pdf", dpi=300, bbox_inches = 'tight')
```


A
Screen dimensions (100, 100)
B
Screen dimensions (200, 200)
D
T
D
D
D
D
T