UNIVERSIDADE FEDERAL DE SANTA CATARINA - CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Desenvolvimento de um Modelo de Avaliação da Originalidade da Paleta de Cores no Design de Interface de Aplicativos Android

Igor Perazzoli

Florianópolis
2022

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Desenvolvimento de um Modelo para Avaliação da Originalidade da Paleta de Cores no Design de Interface de Aplicativos Android

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação, do Departamento de Informática e Estatística, do Centro Tecnológico da Universidade Federal de Santa Catarina, requisito parcial à obtenção do título de Bacharel em Ciências da Computação.

Autor: Igor Perazzoli

Orientadora: Prof.ª Dr.ª rer. nat. Christiane Gresse von Wangenheim, PMP
Coorientadora: Prof.ª Nathalia da Cruz Alves, MSc.

Florianópolis
2022

RESUMO

Atualmente, o desenvolvimento de artefatos e produtos criativos expõe a importância da aprendizagem e desenvolvimento da criatividade e, principalmente, da originalidade. Isso pode ser estimulado desde a Educação Básica por meio do ensino de computação com o desenvolvimento de apps com App Inventor. Uma parte importante desse desenvolvimento é o design de interface, incluindo o design visual e especificamente a paleta de cores do app. Assim, para auxiliar o processo de aprendizagem do aluno é importante fornecer uma avaliação de *feedback* da originalidade da paleta de cores do aplicativo criado por ele de forma automatizada. Nesse contexto, o presente TCC visa desenvolver e testar um modelo capaz de avaliar a originalidade da paleta de cores no design de interface de apps Android desenvolvidos com App Inventor. Para isso, são adotadas técnicas de visão computacional para automatizar a análise da originalidade de cores. O modelo desenvolvido tem como base a fundamentação teórica sobre a originalidade, o design de interfaces de usuário e a visão computacional, além do levantamento do estado da arte relacionado à análise automática da originalidade das paletas de cores em aplicativos. Assim procura-se contribuir para aperfeiçoar a criatividade dos alunos no processo de aprendizagem dentro do ensino de computação na Educação Básica.

Palavras chave: Originalidade, Similaridade, Criatividade, Cores, App Inventor, Design de Interface de Usuário, Educação Básica.

SUMÁRIO

1. INTRODUÇÃO 4
1.1. Contextualização 4
1.2. Objetivos 5
1.3. Metodologia de Pesquisa 6
1.4. Estrutura do Documento 7
2. FUNDAMENTAÇÃO TEÓRICA 8
2.1. Originalidade no contexto da criatividade 8
2.2. Desenvolvimento de apps com App Inventor 11
2.3. Design de interface de apps 15
2.4. Sistemas de Cores 17
2.4.1. Sistema RGB 18
2.4.2. Sistema HSV 19
2.4.3. Sistema CIE 20
2.4.4. Medidas de distância no contexto de comparação de paletas de cores 23
2.4.5 Algoritmo Húngaro 25
3. ESTADO DA ARTE 26
3.1. Definição do Protocolo de Revisão 26
3.2. Execução da Busca 28
3.3. Extração de dados 29
3.4. Análise das abordagens relevantes 29
3.5. Discussão 33
4. PROPOSTA DA SOLUÇÃO 34
4.1. Análise dos requisitos 34
4.2. Modelagem 35
4.2.1 Estudo comparativo da originalidade das cores entre aplicativos 36
4.2.2 Definição do modelo de cores 41
4.3 Definição dos parâmetros da paleta de cores 41
4.3.1 Definição do grau de tolerância 42
4.3.2 Definição do valor limite de threshold de porcentagem 46
4.3.3 Definição da quantidade máxima de cores na paleta 49
4.4 Cálculo da originalidade entre dois aplicativos 51
4.5 Cálculo da nota final de originalidade 56
4.6 Avaliação do cálculo da nota final de originalidade 57
4.7 Implementação 58
4.7.1 Classe “App” 59
4.7.2 Classe “Universe” 62
4.7.3 Classe “Originality” 63
4.4.4 Funções adicionais 67
5. CONCLUSÃO 68
REFERÊNCIAS 70
ANEXO A 73
ANEXO B 84
ANEXO C 90
ANEXO D 100

1. INTRODUÇÃO

1.1. Contextualização

Ao reconhecer a importância do desenvolvimento de artefatos e produtos criativos e originais, observa-se a relevância do processo de aprendizado e desenvolvimento da criatividade, incluindo a característica da originalidade, desde a Educação Básica (Cavallo et al., 2016). E por mais que as áreas artísticas sejam relacionadas frequentemente à criatividade, segundo a Base Nacional Comum Curricular (BNCC) o ensino da computação na escola consegue ser uma ferramenta que também permite estimular a criatividade (Brasil, 2022). Nesse contexto escolar, a criatividade é estimulada principalmente por meio de plataformas de programação em blocos para que os alunos desenvolvam criações computacionais como jogos, aplicativos e animações (Yadav e Cooper, 2017), utilizando *softwares* como o Scratch (https://scratch.mit.edu/) e o App Inventor (https://appinventor.mit.edu).

O App Inventor é um ambiente de programação visual intuitiva que permite construir aplicativos funcionais para sistemas Android (e simular para sistemas iOS). Esses ambientes em blocos inspiram o empoderamento intelectual e criativo (Wolber et al., 2014), especialmente adotando a ação computacional como uma estratégia para levar os alunos a criar soluções reais que podem fazer a diferença, alcançando um impacto social valioso para as comunidades que vivem (Patton et al., 2019).

Uma parte importante do desenvolvimento de um aplicativo é a criação do design de interface do usuário (Solecki, 2020). Assim, como parte do desenvolvimento de um novo aplicativo, o aluno deve definir os elementos que compõem a tela do usuário e o *layout*, além do design visual incluindo a definição de fontes, cores, imagens, etc. Dessa forma, no contexto de estimular a criatividade no desenvolvimento de apps espera-se também o estímulo na criação de design de interface criativos e originais. Sendo que um design de interface é considerado original em termos do design visual, especialmente em relação à paleta de cores. Assim, uma interface de usuário pode ser considerada original em termos da paleta de cores utilizada se ela é diferente das paletas tipicamente utilizadas (Alves et al., 2020). Desta forma, o foco deste trabalho está contido no design de interface do app, especificamente referente à aparência, voltado à paleta de cores.

Para auxiliar o aluno na aprendizagem da criatividade se faz necessária uma forma de retornar um *feedback* do desempenho do aluno (Mishra e Henriksen, 2013). Isto é importante especialmente nessa situação em que o aluno desenvolve um artefato (aplicativo) como resultado da atividade, visto que essa avaliação possui maior

complexidade nos casos em que o escopo não é rigorosamente definido. Isso deve incluir também um *feedback* sobre a originalidade do design de interface de usuário, especificamente sobre a originalidade da paleta de cores.

A avaliação da originalidade da paleta de cores dos aplicativos é complexa. Ao procurar por pesquisas relacionadas constata-se que ainda não há soluções para a avaliação da originalidade da paleta de cores em aplicativos, somente em interfaces *web* e de *desktop*. Sendo que nenhuma destas soluções é desenvolvida em um contexto educacional, mas em contextos profissionais.

Assim, o desafio está em implementar uma forma de medir o grau de originalidade no uso das cores utilizadas de forma confiável e válida dentro do contexto de ensino de computação com App Inventor na Educação Básica. Visando a minimização de esforço e tempo para avaliação, permitindo também um *feedback* em tempo real (Alves, 2019), é importante automatizar a avaliação da originalidade, comparando a escolha das cores com paletas de cores utilizadas em universos de referências por outros alunos em contexto educacionais similares.

Dessa forma, o modelo a ser construído neste trabalho pretende facilitar a avaliação dos aplicativos de alunos por profissionais de educação, que, por vezes, não têm o conhecimento necessário para julgar o quanto uma paleta de cores de uma interface é original, além de reduzir o seu esforço e tempo nessa tarefa.

1.2. Objetivos

Objetivo geral

O objetivo geral deste trabalho é desenvolver e testar um modelo de avaliação da originalidade da paleta de cores no design de interface de apps Android desenvolvidos com App Inventor. São adotadas técnicas de visão computacional para automatizar a análise da originalidade de cores.

Objetivos Específicos

O1. Analisar a fundamentação teórica sobre originalidade, design de interfaces de usuário de apps de Android, e visão computacional.
O2. Analisar o estado da arte em relação à análise automática da originalidade de paletas de cores no design de interfaces de apps.

O3. Desenvolver e testar um modelo de avaliação usando técnicas de visão computacional.

Premissas e restrições

O trabalho é realizado de acordo com o regulamento vigente do Departamento de Informática e Estatística (INE – UFSC) em relação aos Trabalhos de Conclusão de Curso.

O modelo proposto tem como foco a análise e avaliação da originalidade de cores de interfaces não abordando outras dimensões. O presente trabalho também foca na análise de aplicativos Android não abordando interfaces de apps de outras plataformas. A automatização foca em projetos desenvolvidos com a ferramenta de desenvolvimento App Inventor no ensino de computação na educação básica. Esse trabalho aborda somente a avaliação da originalidade da paleta de cores, não incluindo, p.ex., avaliações de grau de harmonia de cores ou alinhamento das escolhas das cores com guias de estilo.

1.3. Metodologia de Pesquisa

A metodologia de pesquisa utilizada neste trabalho é multi-método e dividida nas seguintes etapas.

Etapa 1 – Fundamentação teórica

Estudo, análise e síntese dos conceitos principais e a teoria referente aos temas a serem abordados neste trabalho para a fundamentação teórica utilizando a metodologia de revisão narrativa (Cordeiro et al., 2007). Nesta etapa são realizadas as seguintes atividades:

A1.1 – Análise teórica sobre originalidade no contexto da criatividade

A1.2 – Análise teórica sobre design de interface de apps desenvolvidos com App Inventor

A1.3 – Análise teórica sobre visão computacional

Etapa 2 – Estado da arte

Nesta etapa é realizado um mapeamento sistemático da literatura seguindo o processo proposto por Petersen et al. (2015) para identificar e analisar modelos de análise automatizado da originalidade de design de interfaces de usuário de apps atualmente sendo utilizados. Esta etapa é dividida nas seguintes atividades:

A2.1 – Definição do protocolo da revisão

A2.2 – Execução da busca e seleção de artigos relevantes

A2.3 – Extração e análise de informações relevantes

Etapa 3 – Desenvolvimento

Nesta etapa é desenvolvido um modelo para análise da originalidade de design de interfaces de apps criados no App Inventor, seguindo um processo de modelos de avaliação. Esta etapa é dividida nas seguintes atividades:

A3.1 – Análise de requisitos

A3.2 – Definir o objetivo/critérios de avaliação

A3.3 – Proposta da solução

A3.4 – Identificar a percepção humana das paletas de cores

A3.5 – Implementação da automação da avaliação

A3.6 – Avaliação do modelo em termos de corretude/correlação com julgamento humano

1.4. Estrutura do Documento

No capítulo 2 são apresentadas as fundamentações teóricas sobre a originalidade dentro do contexto da criatividade, o desenvolvimento de aplicativos utilizando o App Inventor, o design de interface do usuário, os sistemas de cores em um panorama geral e algumas métricas que calculam a similaridade das cores. O capítulo 3 descreve o estado da arte por meio de uma revisão bibliográfica, analisando os documentos relevantes selecionados. O capítulo 4 apresenta a proposta da solução, apresentando uma análise de requisitos, a modelagem, o cálculo da originalidade passo a passo, uma avaliação dos parâmetros definidos e a implementação. No capítulo 5 são descritos os resultados obtidos e uma breve discussão sobre.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. Originalidade no contexto da criatividade

A criatividade está atrelada ao processo de criação, responsável por produzir algo novo e valioso, seja pensando a partir de pontos previamente estabelecidos ou sem um processo definido (Alves et al., 2021). A competência da criatividade dentro do contexto educacional é uma parte que requer muita atenção. Por meio dela, alunos se tornam capazes de inventar e criar utilizando suas próprias ideias, trazendo novas contribuições para a sociedade.

Na tentativa de operacionalizar o estudo da criatividade, diferentes pesquisas propuseram maneiras de estruturar a criatividade. Uma destas pesquisas, feita por Rhode (1961), define o que é conhecido como os quatro P's da criatividade (Figura 1):

- *Process* (Processo): Relacionado ao processo mental exercido durante o esforço criativo. Essa parte envolve todo o aprendizado adquirido e comunicação em conjunto com as ferramentas e estratégias utilizadas;
- *Press* (Ambiente): É a ambientação em que a criatividade se forma e sua influência sobre a criatividade dos indivíduos nela situados;
- *Person* (Pessoa): Se trata do indivíduo que está desenvolvendo o ato criativo. Incluindo seus traços de personalidade e atitudes da criatividade individual;
- *Product* (Produto): Se refere ao produto propriamente dito, o resultado do processo criativo. Neste trabalho, os produtos são os aplicativos produzidos pelos alunos.

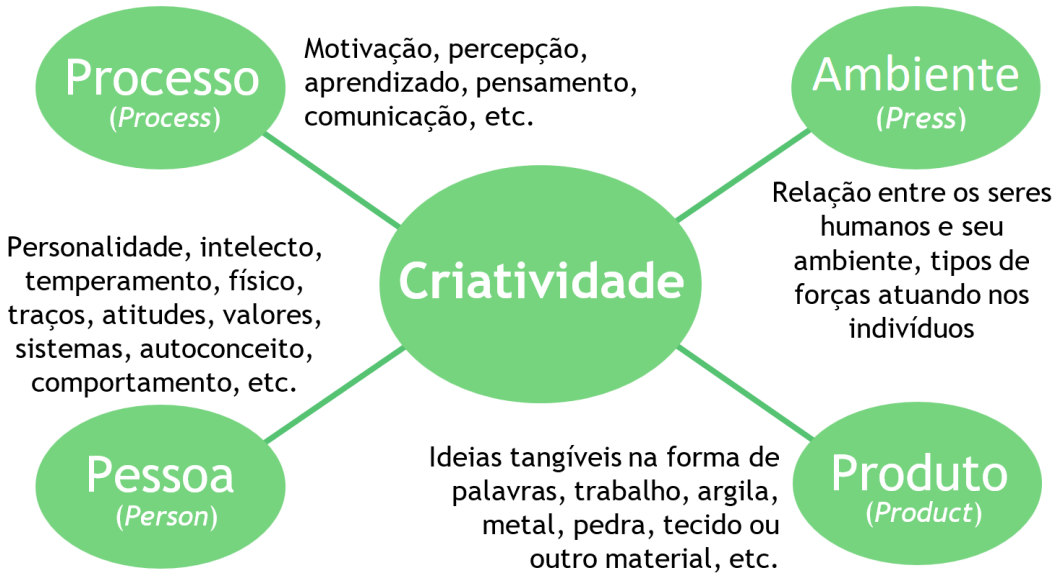


Figura 1. O modelo de estruturação da criatividade dos quatro P's

Fonte: Adaptado de Rhodes (1961)

Mesmo que cada uma das quatro características sejam fundamentais para o entendimento da criatividade, a observação do produto final permite ter uma compreensão da criatividade como um todo.

Uma maneira dos alunos expressarem suas ideias de forma criativa é por meio do desenvolvimento de artefatos computacionais. Um exemplo disso é o desenvolvimento de aplicativos criativos utilizando o ambiente de programação baseado em blocos visuais, como ocorre no App Inventor (Wolber et al., 2014) seguindo o ciclo “Use-Modifique-Crie” (UMC) (Lytle et al., 2019). Adotando essa estratégia, o aprendizado decorre do “uso” de um artefato computacional, da “modificação” de um artefato existente e pela “criação” de um artefato novo. O uso do ciclo Use-Crie-Modifique (UMC) (Figura 2) consiste em desenvolver um aplicativo original em três etapas. Na etapa inicial “Use”, o aluno desenvolve um aplicativo base sem haver uma parte de “criação” própria do aluno. Na etapa seguinte “Modifique”, o aluno deve modificar parte do aplicativo, o tornando “parcialmente seu”. Na etapa final “Crie”, o aluno deve desenvolver seu aplicativo de forma a ser muito diferente dos aplicativos das etapas anteriores, demonstrando um nível superior de originalidade (Cruz Alves et al., 2020).

Figura 2. Etapas do modelo Use-Crie-Modifique

Fonte: Alves et al. (2020)

A criatividade de um produto é uma competência complexa e difícil de ser mensurada. Ela pode ser dividida em características particulares (Alves et al., 2020), como a novidade (“nível de surpresa” do produto), a adequação (quanto um produto atende as necessidades práticas do problema) e a condensação (algumas vezes o produto se torna mais complexo que o design inicial ou pode ser refinado e simplificado).

A novidade é separada em duas áreas principais, a originalidade e a mudança de paradigma (Alves et al., 2020). De maneira abrangente, uma ideia ou um produto é considerado original se apresenta algo inovador no mercado, e apesar da originalidade não ser suficiente para designar um produto como sendo criativo, ela serve como um indicativo fundamental para avaliar uma produção criativa (Mishra e Henriksen, 2013). Sabendo que dentro do contexto educacional não se espera que os alunos alcancem um nível tão avançado de desenvolvimento da criatividade, é desconsiderada a mudança de paradigma do produto gerado.

Assim, a avaliação de um artefato computacional pode considerar várias camadas, seja considerando o objetivo, o escopo, o design ou o código (Alves et al., 2020). Essas camadas para avaliar um produto (Figura 3) representam as diferentes visões aplicadas para avaliar um aplicativo como original. Dentro do contexto deste trabalho, o foco é avaliar a originalidade da paleta de cores, uma parte do elemento de aparência contido na camada de design do modelo utilizado.

Para que o aluno tenha ciência do seu desenvolvimento da criatividade, é necessário avaliar sua produção criativa dando um retorno construtivo para que ele se sinta estimulado a continuar aperfeiçoando sua criatividade (Mishra e Henriksen, 2013). A originalidade dos aplicativos pode ser representada em diferentes camadas (Figura 3) (Alves et al., 2020). Isso também inclui a originalidade do design, mais especificamente a etapa da aparência, na qual a originalidade da paleta de cores está inserida.

Figura 3. Camadas de avaliação da originalidade de apps

Fonte: Adaptado de Alves et al. (2020)

2.2. Desenvolvimento de apps com App Inventor

Sendo um ambiente de programação visual em blocos feito especificamente para a criação de aplicativos móveis para dispositivos com o sistema Android, o App Inventor (MIT, 2022) pode ser acessado pelo navegador web de forma gratuita por qualquer pessoa. Com mais de um milhão de visitantes mensais de 195 países diferentes, produzindo quase 30 milhões de aplicativos (MIT, 2022), o App Inventor possibilita que pessoas com diferentes níveis de conhecimento sejam capazes de desenvolver aplicações. Isso é alcançado pela alta acessibilidade difundida por meio da programação via blocos, algo que dispensa a necessidade de compreensão do código e sua sintaxe (CSTA, 2017).

Um aplicativo desenvolvido no App Inventor é dividido em duas partes principais, o designer e o desenvolvimento através dos blocos.

Na aba Designer (Figura 4) é possível encontrar as configurações e ferramentas necessárias para inserir os mais diferentes componentes ao esqueleto da interface, sendo que cada elemento pode possuir uma configuração visual própria, alterando detalhes como a proporção e a cor, por exemplo.

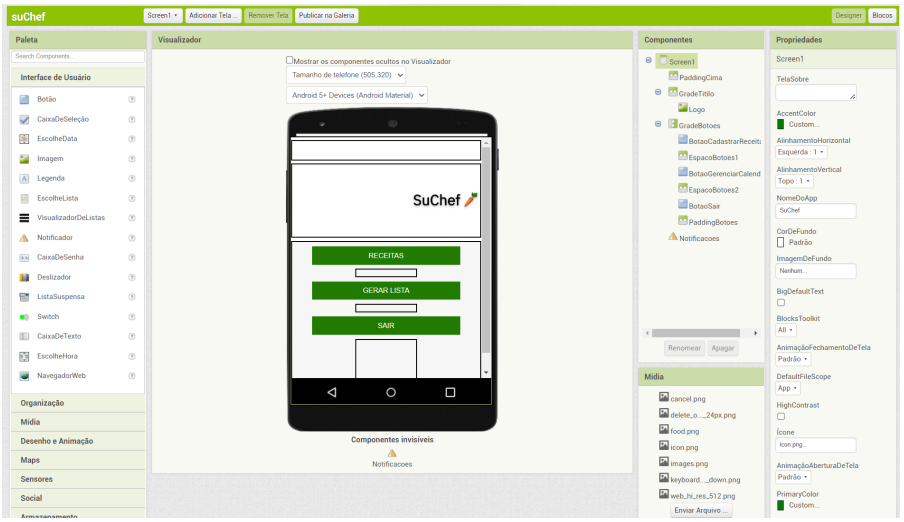


Figura 4. Aba "Designer" dentro do App Inventor

Fonte: Adaptado de MIT (2022)

Já a parte do desenvolvimento pelos blocos visuais (Figura 5) contém toda a programação lógica da aplicação. Definindo interações por meio de blocos de funções, laços, definição de variáveis, entre outros. O contexto da lógica dos blocos depende dos componentes contidos no aplicativo, pois toda interação ocorre por intermédio desses elementos.

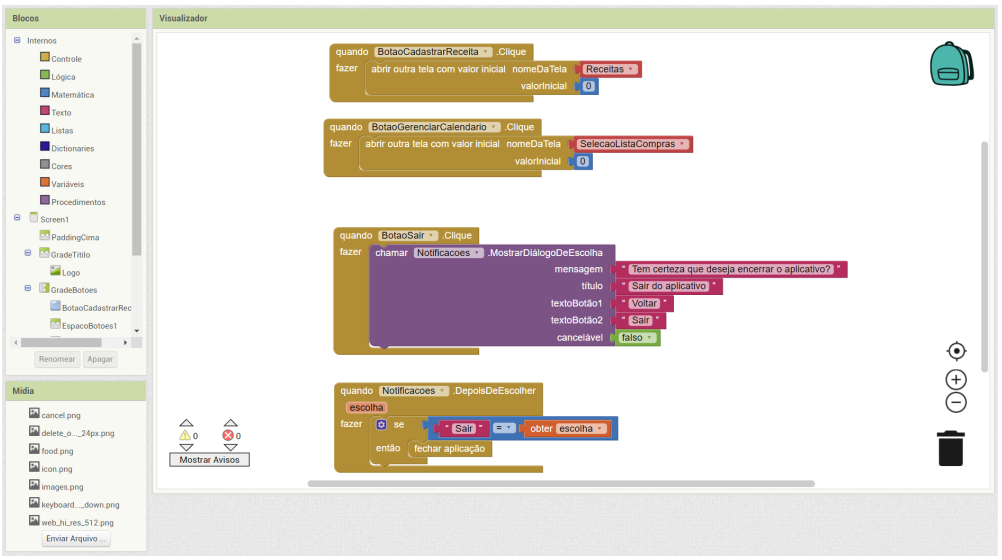


Figura 5. Aba "Blocos" dentro do App Inventor

Fonte: Adaptado de MIT (2022)

O App Inventor possui uma ampla variedade de componentes de interface que possuem propriedades configuráveis de cores (Tabela 1).

Componente	Descrição	Propriedades de Cores
Tela	Componente que comporta todos os demais componentes, sendo que um aplicativo pode possuir mais de uma tela	Cor de destaque, Cor do fundo, Cor primária, Cor primária escura
Botão	Componente com capacidade de detectar cliques	Cor do fundo, Cor do texto
CaixaDeSeleção	Componente capaz de disparar um evento quando o usuário clica nele	Cor do fundo, Cor do texto
EscolheData	Ao clicar nesse botão exibe um diálogo para o usuário escolher uma data	Cor do fundo, Cor do texto
Legenda	Exibe um texto	Cor do fundo, Cor do texto
EscolheLista	Ao clicar nesse botão exibe uma lista de textos para o usuário escolher	Cor do fundo, Cor do texto, Cor do fundo do item, Cor do texto do item
VisualizadorDeListas	Componente visível que permite colocar uma lista de elementos de texto para apresentação na tela	Cor do fundo, Cor da seleção, Cor do texto, Cor do detalhe do texto
Notificador	Componente que exibe diálogos de alerta, mensagens e alertas temporários, além de criar entradas de log Android	Cor do fundo, Cor do texto
CaixaDeSenha	Uma caixa de texto que oculta o conteúdo inserido	Cor do fundo, Cor do texto
Deslizador	Uma barra de progresso que adiciona um indicador arrastável, sendo que o indicador pode ser arrastado para a esquerda ou para direita para ajustar sua posição	Cor à esquerda, Cor à direita
Interruptor (*Switch*)	Gera um evento ao ser pressionado pelo usuário	Cor do fundo, Cor do texto, Cor do botão ativo, Cor do botão inativo, Cor do traço ativo, Cor do traço inativo
CaixaDeTexto	Componente que permite que o usuário insira um texto	Cor do fundo, Cor do texto

EscolheHora	Um botão que, quando clicado, inicia um diálogo que permite ao usuário selecionar um horário	Cor do fundo, Cor do texto
OrganizaçãoHorizontal	Dispõe outros componentes linearmente, da esquerda para a direita	Cor do fundo
OrganizaçãoDeRolagemHorizontal	Similar a organização horizontal, com a adição da barra de rolagem	Cor do fundo
OrganizaçãoVertical	Dispõe outros componentes linearmente, de cima para baixo, alinhados à esquerda	Cor do fundo
OrganizaçãoDeRolagemVertical	Similar a organização vertical, com a adição da barra de rolagem	Cor do fundo
EscolheImagem	Quando pressionado, exibe a galeria de imagens do dispositivo para que o usuário escolha uma imagem	Cor do fundo, Cor do texto
Bola	Um *sprite* redondo que pode ser inserido em uma pintura e capaz de reagir a diferentes interações	Cor da pintura
Pintura	Um painel retangular no qual o desenho pode ser feito e os *sprites* podem ser movidos	Cor do fundo, Cor de pintura
Círculo (Mapa)	Adiciona um círculo ao mapa	Cor de preenchimento, Cor do traço
Linha (Mapa)	Adiciona uma linha ao mapa	Cor do traço
Marcador (Mapa)	Adiciona um marcador ao mapa	Cor de preenchimento, Cor do traço
Polígono (Mapa)	Adiciona um polígono ao mapa	Cor de preenchimento, Cor do traço
Retângulo (Mapa)	Adiciona um retângulo ao mapa	Cor de preenchimento, Cor do traço
EscolheContato	Um botão que, ao ser clicado, exibe uma lista de contatos para o usuário escolher	Cor do fundo, Cor do texto
EscolheEmail	Uma caixa de texto que sugere contatos quando o usuário começa a digitar um nome ou e-mail	Cor do fundo, Cor do texto
EscolheNúmeroDeTelefone	Um botão que, quando clicado, mostra uma lista dos números de telefone dos contatos para que o usuário escolha um	Cor do fundo, Cor do texto

Tabela 1. Componentes de design de interface de usuário no App Inventor com propriedades de cores

Fonte: Adaptado de MIT (2022)

Esses componentes de interface de usuário (Tabela 1) podem ser configurados em termos de cores em relação a vários aspectos (Tabela 2).

Propriedade	Descrição
Cor de destaque	Define a cor usada para realçar componentes como o notificador e o seletor de data
Cor do fundo	Define a cor da base interna do componente
Cor primária	Define a cor usada como parte do tema do sistema operacional, incluindo a barra de título
Cor primária escura	Define a cor usada como parte do tema do sistema operacional, incluindo a barra de título, quando o tema está definido para “escuro”
Cor do texto	Define a cor base a ser usada nos componentes de texto
Cor do fundo do item	Define a cor da base interna dos itens de um seletor de lista
Cor do texto do item	Define a cor do texto dos itens contidos em um seletor de lista

Cor à esquerda	Define a cor da esquerda da barra interna de um deslizador
Cor à direita	Define a cor da direita da barra interna de um deslizador
Cor do botão ativo	Define a cor do botão quando o interruptor (*switch*) está no estado ligado
Cor do botão inativo	Define a cor do botão quando o interruptor (*switch*) está no estado desligado
Cor do traço ativo	Define a cor do traço quando o interruptor (*switch*) está no estado ligado
Cor do traço inativo	Define a cor do traço quando o interruptor (*switch*) está no estado desligado
Cor da pintura	Define a cor base utilizada no componente da pintura
Cor de preenchimento	Define a cor usada para preencher os componentes do mapa
Cor do traço	Define a cor usada nos contornos dos componentes do mapa

Tabela 2. Funções das propriedades de cores no App Inventor

Fonte: Adaptado de MIT (2022)

Como apresentado na Figura 6, a gama de cores iniciais disponibilizadas pelo App Inventor são cores geralmente com alta saturação, com exceção das cores neutras (branco, preto e cinza). No entanto, é possível customizar a cor do componente ao selecionar a opção "*Custom*", por meio de um pop-up com a opção de escolha visual da cor ou inserindo o código do padrão de cores HEX para selecionar uma cor diferente das opções padrões. Além disso, o pop-up fornece as possibilidades de customização da opacidade e luminosidade da cor selecionada por meio de duas barras laterais.

Figura 6. Opções de configuração de cores da aba "Designer" dentro do App Inventor

Fonte: Adaptado de MIT (2022)

Outra forma de alterar as cores no App Inventor é por meio da aba "Blocos" (Figura 7). As cores podem ser predefinidas em um bloco por um seletor de cores mais limitado ou podem ter um valor definido por três valores inteiros (variando de 0 até 255 cada) representando uma cor do sistema de cores RGB.

Figura 7. Opções de configuração de cores na aba "Blocos" dentro do App Inventor

Fonte: Adaptado de MIT (2022)

Os aplicativos desenvolvidos no App Inventor são salvos automaticamente na nuvem. Existe também a possibilidade de exportar o projeto no formato ".aia". Esse formato possui um conjunto de arquivos compactados que incluem arquivos de propriedade do projeto, arquivos de mídia usados pelo aplicativo e, para cada tela do aplicativo, um arquivo ".bky" e um arquivo ".scm". O arquivo ".scm" contém uma representação JSON de seus componentes de interface de usuário enquanto o arquivo ".bky" uma representação XML com os blocos de programação usados na lógica do aplicativo.

2.3. Design de interface de apps

O design visual é uma parte importante do design de interface de usuário. Ela se refere aos elementos visuais do aplicativo, utilizados para manter a atratividade/estética visual e usabilidade (Solecki, 2020). Durante o desenvolvimento do design visual algumas características devem ser levadas em consideração (Schlatter e Levinson, 2013):

- Consistência: definir um consenso quanto ao posicionamento e tratamento dos componentes para manter o design visual compreensível ao usuário;
- Hierarquia: a distribuição dos elementos deve ser feita de forma hierárquica, destacando os elementos considerados mais importantes para aumentar a atratividade;
- Personalidade: adição de composições e efeitos que remetem ao aplicativo em questão.

A harmonia entre essas características pode assegurar um aplicativo organizado e agradável ao olhar do usuário (Solecki, 2020).

Como parte fundamental da interface do usuário, o design visual é composto por elementos (cores, fontes, imagens/ícones) que definem a aparência da aplicação, determinando como a informação é apresentada visualmente na tela.

Um elemento importante do design visual são as cores, que evocam muitas emoções diferentes e influenciam a estética em geral (Lima et al., 2021). As cores são essenciais para compor um ambiente atrativo ao usuário, ajudando a identificar o que está relacionado e indicando os estados dos componentes (Schlatter e Levinson, 2013). A paleta de cores pode ser compreendida como uma forma de representar um conjunto de cores utilizadas no design de interface de um aplicativo. Seguindo as diretrizes do Material Design (Google, 2019) a paleta de cores de um aplicativo deve ser composta de uma cor primária e uma cor secundária, além dos tons de branco, cinza e preto. Porém, observa-se que muitos apps desenvolvidos com App Inventor usam um número bem maior de cores diferentes (Figura 8).

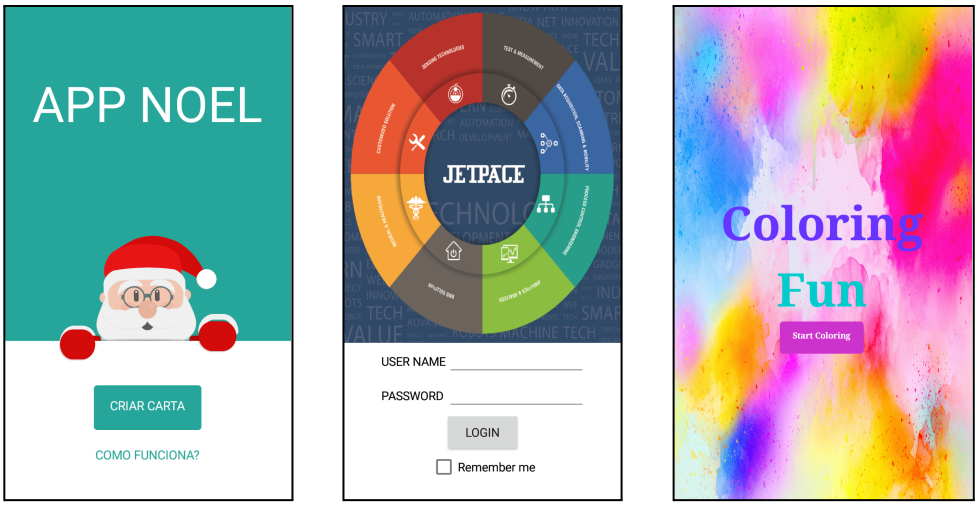


Figura 8. Exemplos de aplicativos desenvolvidos com App Inventor

Fonte: Autor

Um aspecto importante do esquema de cores utilizado é a sua composição harmônica (Figura 9) utilizando a teoria das cores do círculo cromático. Essa representação facilita no momento de escolha das cores que irão compor o aplicativo.

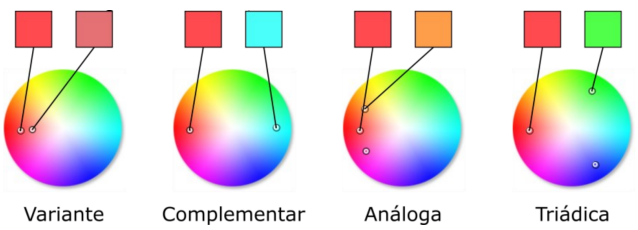


Figura 9. Composição harmônica das cores

Fonte: Adaptado de Sessions College (2019)

A distribuição das cores na interface de um app está presente em elementos como o fundo, o texto, os botões, as bordas, entre outros.

Tão importante quanto as cores dos elementos são as cores das imagens (Solecki, 2020). Compostas por muitos pixels, as imagens possuem usos variados (Figura 10), indo das mais simples, como uma indicação para voltar uma tela, até as mais detalhadas, como uma animação. Dentro das imagens simplificadas, há os ícones que representam ações, objetos e conceitos da interface.

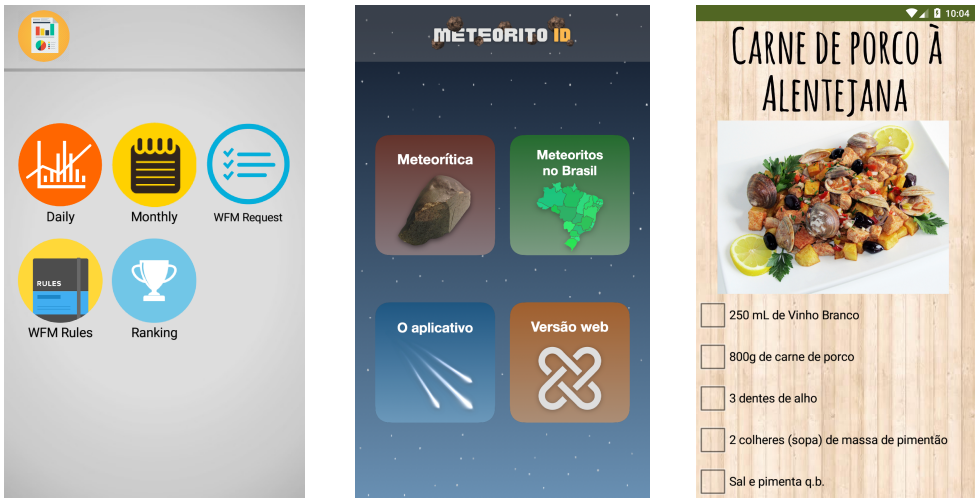


Figura 10. Aplicativos desenvolvidos com App Inventor contendo imagens e ícones

Fonte: Autor

Considerando essa importância do design visual, é perceptível que essa parte também deve entrar no ambiente educacional para contribuir com o amadurecimento da criatividade dos alunos, acompanhando, avaliando e fornecendo um *feedback* sobre os aplicativos criados.

2.4. Sistemas de Cores

Um sistema de cor pode ser definido como uma representação digital das cores percebidas pelo ser humano (Ibraheem et al., 2012), sendo que as cores são obtidas por um processo de combinação de diferentes valores de um conjunto de cores primárias. Tipicamente, os sistemas de cores possuem três ou quatro componentes de cores. A existência de diferentes sistemas de cores é justificada pelas diferentes necessidades que cada um comporta. Os sistemas de cores são divididos em três categorias (Ibraheem et al., 2012) de acordo com suas aplicabilidades:

- Orientados a dispositivos: São os sistemas dependentes de dispositivos que se relacionam e são afetados pelo sinal do dispositivo. A cor resultante é afetada pelas ferramentas usadas para a exibição. Aqui se encontram dispositivos de hardware usados para a percepção da visão humana, como TV e sistemas de vídeo;
- Orientados ao usuário: Considerados um caminho entre o observador e o dispositivo que trata as informações de cor. Esses sistemas permitem que o usuário descreva e aproxime o que percebe visualmente ao apresentar a cor;
- Independentes do dispositivo: Esses sistemas não são afetados pelas propriedades do dispositivo. A cor resultante é a mesma independente do desempenho do dispositivo. Úteis para informações de transmissão de rede para que os dados visuais passem por diferentes dispositivos.

Os principais sistemas de cores incluem o sistema RGB, popularmente usado em dispositivos eletrônicos (Ibraheem et al., 2012), o sistema HSV, que representa efetivamente as cores em computadores gráficos (Chen et al., 2007) e o sistema CIE L*a*b, que é um dos sistemas mais próximos da percepção humana (Ibraheem et al., 2012).

2.4.1. Sistema RGB

Baseado nas três cores primárias vermelho (*Red*), verde (*Green*) e azul (*Blue*), o modelo RGB (Figura 11) apresenta um espectro de luz, combinando diferentes cores para formar uma nova cor (Loesdau et al., 2014), a qual também pode ser usada para gerar uma nova gama de cores.

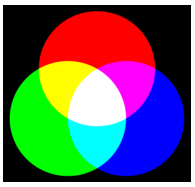


Figura 11. Diagrama do sistema de cores RGB

Fonte: Wikipedia

O sistema RGB pode ser representado na forma de um cubo (Figura 12) pelos valores de cores no alcance de [0,1] (Ibraheem et al., 2012), tendo os valores das cores cinzas na diagonal principal formada pelos pontos do valor da cor preta (0,0,0) e do valor da cor branca (1,1,1). Este sistema é amplamente utilizado para a reprodução visual de imagens, visto que não é necessário mais transformações para a exibição na tela.

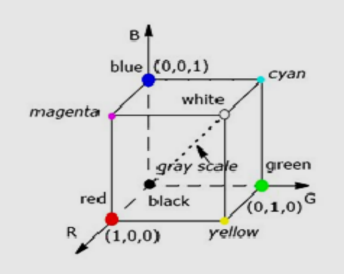


Figura 12. Representação das cores do sistema RGB em um cubo

Fonte: Ibraheem et al. (2012)

Há duas classificações para o sistema RGB, o linear e o não-linear (Ibraheem et al., 2012). O sistema RGB linear é raramente utilizado para representar imagens (usado mais para aplicações de computação gráfica), pelo fato de não ser adequado para a análise numérica. Por outro lado, o sistema RGB não-linear é a escolha geral para a visualização de imagens, em que a captura de uma câmera, por exemplo, gera valores de entrada representados na faixa de 0 a 255. Esses dados são armazenados para posterior uso em aplicações de processamento de imagem.

Outra forma de representar uma cor do sistema RGB é por meio de um código hexadecimal de seis dígitos, sendo que um byte equivale tanto a dois dígitos hexadecimais (variando de 00 até FF), quanto a notação decimal de 0 a 255 mencionada anteriormente. Por exemplo, uma cor que possui os parâmetros vermelho = 3, verde = 252 e azul = 161, pode ser convertida para os hexadecimais 03, FC e A1 respectivamente, compondo o código #03FCA1.

O modelo RGB é amplamente utilizado em dispositivos emissores de luzes (Ibraheem et al., 2012), como monitores e telas de celulares, pois o nosso cérebro interpreta os comprimentos de onda nas cores vermelho, verde e azul. A similaridade no sistema RGB é facilmente calculada por métodos genéricos (como a distância Euclidiana), porém seu resultado nem sempre é preciso devido à percepção humana da similaridade não ser muito próxima do valor gerado (Khoker et al. 2016).

2.4.2. Sistema HSV

O modelo HSV (*Hue*, *Saturation* e *Value*) é originado pela família de modelos HSI (*Hue*, *Saturation* e *Intensity*), com a característica fundamental que o componente de intensidade é separado dos componentes cromáticos de matiz e saturação (Chen et al., 2007), pois estes componentes dependem da percepção humana do espectro de cores.

O diferencial do HSV (Figura 13) está no uso do elemento de valor do brilho, utilizado no lugar do valor da intensidade. Esse sistema considera que a percepção

humana, quando vendo uma imagem, se caracteriza inicialmente pelo brilho e cromaticidade, para só depois ser definido pela matiz e saturação (Chen et al., 2007). O brilho é tido como uma medida da intensidade luminosa dada em graus, a matiz é o atributo que representa a cor dominante em porcentagem e a saturação é uma expressão do grau que uma cor pura é diluída pela luz branca, também dada em porcentagem.

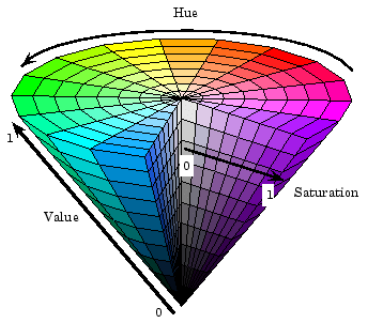


Figura 13. Diagrama do sistema de cores HSV

Fonte: Ibraheem et al. (2012)

Estes atributos extraídos do modelo HSV representam adequadamente as diferentes características dos gráficos do computador, sendo que os gráficos do computador possuem menos cores.

Para medir a similaridade das cores em um sistema HSV frequentemente são utilizadas métricas de distância (como a distância Euclidiana e a distância de Manhattan), como é possível ver em Pardede et al. (2017) e Khoker et al. (2016).

2.4.3. Sistema CIE

O modelo de cores CIE, criado pela Comissão Internacional de Iluminação conhecida como Comissão Internacional de l'Elcairage (CIE) (http://cie.co.at), foi feito manualmente por meio da capacidade de julgamento humano de visualização e correspondência de aparências, ou seja, sua colorimetria é baseada na percepção humana das cores.

O CIELAB (Figura 14), também referenciado como L*a*b*, foi publicado em 1976 (Ibraheem et al., 2012) e desde então é utilizado amplamente para controlar a aceitabilidade das cores. O sistema CIELAB descreve e ordena as cores baseado na teoria oposta da visão de cores (Datacolor, 2013). Essa teoria aponta que algumas combinações de cores, como o vermelho e o verde, não podem ser percebidas ao mesmo tempo, mas sim as combinações: vermelho e amarelo, vermelho e azul, verde e amarelo ou verde e azul.

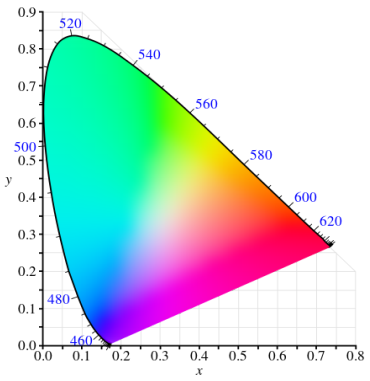


Figura 14. Diagrama do sistema de cores CIE

Fonte: Programming Design Systems (2022)

As siglas L*a*b* (Figura 15) se referem às coordenadas de cores dentro desse sistema (Ibraheem et al., 2012). Sendo que "L*" representa a coordenada de luminosidade, "a" representa a coordenada de verde e vermelho (em que "+a" indica a cor vermelha e "-a" indica a cor verde) e "b" representa a coordenada de amarelo e azul (em que "+b" indica a cor amarela e "-b" indica a cor azul).

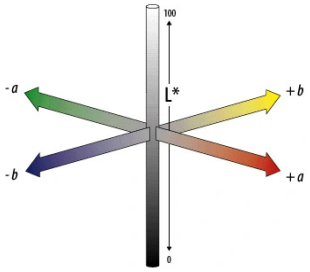


Figura 15. Representação gráfica do sistema CIELAB

Fonte: Ibraheem et al. (2012)

Para definir a similaridade entre duas cores do sistema CIELAB é usado o ΔE (Datacolor, 2013), uma medida da mudança na percepção visual a partir de duas cores. Em uma escala de percepção, a variação de ΔE está dentro do intervalo [0, 100], como é possível visualizar pela Tabela 3.

ΔE	Nível de percepção
Menor ou igual a 1	Não é perceptível pelos olhos humanos
De 1 até 2	Perceptível por meio de uma observação próxima
De 2 até 10	Perceptível ao olhar
De 11 até 49	As cores são mais parecidas que opostas
100	As cores são exatamente opostas

Tabela 3. Significados dos valores de ΔE em relação ao nível de percepção humana

Fonte: Adaptado de Schuessler (2020)

Para calcular o ΔE (Figura 16), seria suficiente apenas medir a distância entre os dois pontos no espaço tridimensional do sistema.

$$\Delta E^*_{ab} = \sqrt{(L^*_2 - L^*_1)^2 + (a^*_2 - a^*_1)^2 + (b^*_2 - b^*_1)^2}$$

Figura 16. Equação de 1976 que define ΔE

Fonte: Schuessler (2020)

Essa solução apesar de satisfatória apresenta um problema com o nível de saturação (Schuessler, 2020). Em situações em que há duas cores com níveis de saturação diferentes, o resultado obtido pode não ser o esperado. Foi a partir disso que melhorias mais complexas surgiram para essa fórmula, dando origem às novas fórmulas de 1994 e, posteriormente, a de 2000.

$$\Delta E^*_{00} = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \frac{\Delta C'}{k_C S_C}\frac{\Delta H'}{k_H S_H}}$$

$$\Delta L' = L^*_2 - L^*_1$$
$$\bar{L} = \frac{L^*_1 + L^*_2}{2} \quad \bar{C} = \frac{C^*_1 + C^*_2}{2}$$
$$a'_1 = a^*_1 + \frac{a^*_1}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right) \quad a'_2 = a^*_2 + \frac{a^*_2}{2}\left(1 - \sqrt{\frac{\bar{C}^7}{\bar{C}^7 + 25^7}}\right)$$
$$\bar{C}' = \frac{C'_1 + C'_2}{2} \text{ and } \Delta C' = C'_2 - C'_1 \quad \text{where } C'_1 = \sqrt{a'^2_1 + b^{*2}_1} \quad C'_2 = \sqrt{a'^2_2 + b^{*2}_2}$$
$$h'_1 = \operatorname{atan2}(b^*_1, a'_1) \mod 360°, \quad h'_2 = \operatorname{atan2}(b^*_2, a'_2) \mod 360°$$

$$\Delta h' = \begin{cases} h'_2 - h'_1 & |h'_1 - h'_2| \le 180° \\ h'_2 - h'_1 + 360° & |h'_1 - h'_2| > 180°, h'_2 \le h'_1 \\ h'_2 - h'_1 - 360° & |h'_1 - h'_2| > 180°, h'_2 > h'_1 \end{cases}$$

$$\Delta H' = 2\sqrt{C'_1 C'_2}\sin(\Delta h'/2), \quad \bar{H}' = \begin{cases} (h'_1 + h'_2 + 360°)/2 & |h'_1 - h'_2| > 180° \\ (h'_1 + h'_2)/2 & |h'_1 - h'_2| \le 180° \end{cases}$$

$$T = 1 - 0.17\cos(\bar{H}' - 30°) + 0.24\cos(2\bar{H}') + 0.32\cos(3\bar{H}' + 6°) - 0.20\cos(4\bar{H}' - 63°)$$
$$S_L = 1 + \frac{0.015\left(\bar{L} - 50\right)^2}{\sqrt{20 + \left(\bar{L} - 50\right)^2}} \quad S_C = 1 + 0.045\bar{C}' \quad S_H = 1 + 0.015\bar{C}'T$$
$$R_T = -2\sqrt{\frac{\bar{C}'^7}{\bar{C}'^7 + 25^7}}\sin\left[60° \cdot \exp\left(-\left[\frac{\bar{H}' - 275°}{25°}\right]^2\right)\right]$$

Figura 17. Equação de 2000 que define ΔE

Fonte: Schuessler (2020)

A equação de 2000 (Figura 17) é a mais precisa até o momento para representar a percepção humana das cores, relacionando corretamente as imprecisões de luminosidade contidas nas equações anteriores (Schuessler, 2020). Isso faz com que essa equação calcule apropriadamente as diferenças de saturação entre as cores.

2.4.4. Medidas de distância no contexto de comparação de paletas de cores

Uma outra forma de calcular a similaridade entre as cores de um sistema, é por meio de cálculos de suas distâncias.

Medir a originalidade utilizando medidas de similaridade é diferente de medir utilizando medidas de distância. Enquanto uma medida de similaridade avalia o quão próximas duas cores são dentro de um determinado sistema de cores, a medida de distância calcula o quão longe estão duas cores dentro desse sistema, para assim avaliar a originalidade. Na questão da semântica, a similaridade é o inverso da originalidade. Duas cores são ditas similares se elas são semelhantes e, portanto, quanto menos similares duas cores são, maior o grau de originalidade entre elas.

Alguns cálculos de distância são genéricos, utilizados para os mais diversos objetivos (como a distância Euclidiana), enquanto que outros foram desenvolvidos especificamente para calcular a distância entre as cores (como a distância d_1).

Distância Euclidiana

A ideia por trás do cálculo da distância euclidiana (Figura 18) se dá pela repetição da aplicação do teorema de Pitágoras, resultando na menor distância entre dois pontos (Zenggang et al., 2021). Sendo que p_i e q_i representam valores contidos em dois conjuntos P e Q, respectivamente, de pontos distintos de dimensão n.

$$d(P,Q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Figura 18. Equação da Distância Euclidiana

Fonte: Adaptado de Zenggang et al. (2021)

***Earth Mover's Distance* (*EMD*)**

A EMD (Figura 19) é uma função que calcula a diferença entre dois histogramas (Tang et al., 2014). Dados dois histogramas P e Q, cada um com o total de n caixas, uma matriz de fluxo F, em que $f_{i,j}$ indica o fluxo necessário para mover de p_i para q_j, e uma matriz de custo C, em que $c_{i,j}$ modela o custo de mover o fluxo da i-ésima caixa para a j-ésima caixa, o cálculo da EMD pode ser feita da seguinte forma:

$$d(P,Q) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{i,j} c_{i,j}$$

Figura 19. Equação utilizada para calcular a EMD

Fonte: Adaptado de Tang et al. (2014)

Assumindo que P e Q estão normalizadas, tal que o somatório de P é igual ao somatório de Q, a EMD pode ser definida formalmente (Figura 20).

$$emd(p,q) = min\, d(p,q),$$
$$\text{tal que } \forall i,\ j \in [1,n] : f_{i,j} \geq 0,$$
$$\forall i \in [1,n] : \sum_{j=1}^{n} f_{i,j} = p_i,$$
$$\text{e } \forall j \in [1,n] : \sum_{i=1}^{n} f_{i,j} = q_j$$

Figura 20. Definição formal da EMD

Fonte: Adaptado de Tang et al. (2014)

Distância de Manhattan (*City Block Distance*)

Enquanto a distância euclidiana faz uso da hipotenusa para calcular a menor distância entre dois pontos, a Distância de Manhattan (Figura 21) calcula a distância por meio da soma entre x e y (em um espaço bidimensional), de forma similar ao modo que as pessoas andam pelas ruas de uma cidade, onde os prédios não permitem um caminho direto de um ponto ao outro (Jyoti et al., 2014). Dados dois pontos a e b de dimensão n, o cálculo da Distância de Manhattan é dada por:

$$d(a,b) = \sum_{i=1}^{n} |a_i - b_i|$$

Figura 21. Equação da Distância de Manhattan

Fonte: Adaptado de Jyoti et al. (2014)

Estatística Qui-quadrado (χ^2-*statistic*)

Dados dois pontos P = {p_1, …, p_n} e Q = {q_1, …, q_n} de dimensão n, a estatística χ^2 (Khokher et al., 2016) calcula a distância entre tais pontos usando a fórmula:

$$d(P,Q) = \sum_{i=1}^{n} \frac{(p_i - m_i)^2}{m_i}$$
$$\text{tal que } m_i = \frac{(p_i + q_i)}{2}$$

Figura 22. Equação da estatística χ^2

Fonte: Adaptado de Khokher et al. (2016)

Distância d_1 (*d_1 distance*)

Assumindo dois vetores P = {p_1, …, p_n} e Q = {q_1, …, q_n} de dimensão n, a distância d_1 (Agarwal et al., 2012) pode ser utilizada para calcular a similaridade das cores por meio da seguinte fórmula:

$$d(P,Q) = \sum_{i=1}^{n} \frac{|p_i - q_i|}{|1 + p_i + q_i|}$$

Figura 23. Equação da distância d_1

Fonte: Adaptado de Agarwal et al. (2012)

Entretanto, vale mencionar que essa equação (Figura 23) foi desenvolvida essencialmente para realizar a comparação da similaridade entre imagens. Sendo que p_i é o vetor de recursos da imagem de consulta, q_i é o vetor de recursos das imagens do banco de dados e n é o tamanho dos vetores.

2.4.5 Algoritmo Húngaro

O algoritmo (ou método) húngaro busca encontrar o custo mínimo de ações para diferentes indivíduos (Hamuda, et al., 2018). Esse algoritmo utiliza uma programação linear para automatizar seus cálculos. A entrada do algoritmo é dada por uma matriz composta de linhas em função das colunas. Nessa matriz, procura-se o valor mínimo de cada linha, atribuindo a linha que possui esse valor para a coluna correspondente, formando um par.

Caso se encontre um valor mínimo em que a coluna já pertença a outro par e esse valor mínimo é melhor que o desse outro par, então um novo par é feito e a linha que ficou sem uma coluna correspondente, busca uma nova coluna para formar seu par, ou seja, busca o segundo valor mínimo daquela linha.

Esse processo é feito até que todos os pares sejam formados. Assim, como resultado, uma matriz com os melhores pares, é formada.

3. ESTADO DA ARTE

O estado da arte é analisado por meio de um mapeamento sistemático que identifica os principais artigos e trabalhos desenvolvidos relacionados com a temática da originalidade de cores em *softwares*.

3.1. Definição do Protocolo de Revisão

O objetivo deste mapeamento é responder a seguinte pergunta de pesquisa: **Quais abordagens existem para automaticamente analisar a originalidade de paletas de cores de aplicativos (com App Inventor)?**

Esta pergunta de pesquisa é refinada nas seguintes questões de análise:

AQ1. Quais abordagens existem para a avaliação da originalidade da paleta de cores de aplicativos e quais suas características?
AQ2. Qual modelo de cores é utilizado para avaliar a originalidade da paleta de cores dos aplicativos?
AQ3. Quais técnicas são adotadas na análise de originalidade/similaridade de cores?
AQ4. Como a qualidade das abordagens foi avaliada e quais as principais descobertas?

Considerando o foco da pesquisa, são definidos os seguintes critérios de inclusão e exclusão:

Critérios de inclusão:

- São incluídos apenas artigos em inglês;
- São considerados somente artigos que apresentam abordagens para a extração automatizada de recursos com base no código ou imagens de capturas de tela (*screenshots*);
- Soluções voltadas a interfaces gráficas de usuários de sistemas de software incluindo sistemas web e aplicativos para dispositivos móveis;

Critérios de exclusão:

- Artigos que não sejam baseados em código ou imagens de screenshots;
- São excluídas interfaces não gráficas como realidade virtual e/ou para domínios específicos como saúde, e/ou para tipo de informações específicas como reprodutores de vídeo.

- São considerados apenas pesquisas publicadas nos últimos 10 anos (desde 2012), levando em consideração o avanço recente especificamente em relação a aplicativos móveis.

Fontes: São pesquisados nos principais bancos de dados e bibliotecas digitais no campo da computação, incluindo as Bibliotecas Digitais ACM, a IEEE Xplore e Scopus com acesso por meio do Portal Capes. A pesquisa também foi feita via Google Scholar, para complementar a busca (Haddaway et. al., 2015, Piasecki et al. 2017).

Critérios de qualidade: São considerados apenas artigos que apresentam informações substanciais para se extrair informações referente às perguntas de análise.

Termos de busca: Com base na questão de pesquisa, várias pesquisas informais foram realizadas para calibrar o *string* de busca, identificando termos de pesquisa relevantes e seus sinônimos (Tabela 4). Foram utilizados sinônimos para minimizar o risco de omitir trabalhos relevantes. Com base em buscas informais para calibrar a *string* de busca, não foram encontrados resultados voltados especificamente a aplicativos móveis, porém alguns resultados potencialmente relevantes de forma geral para sistemas de *software*. Consequentemente foi decidido abrir o escopo da busca para qualquer sistema de *software* com a intenção de encontrar abordagens relacionadas a essa pesquisa.

Termo	**Sinônimos**	**Tradução (inglês)**
Originalidade	originalidade, novidade, similaridade	novelty, originality, similarity
Paleta de cores	cores	“color palette”, colors
Interface de usuário	GUI, “design visual”	“user interface”, GUI, “visual design”

Tabela 4. Termos de busca

***String* de busca:** Após a definição dos termos e seus sinônimos, definiu-se o *string* de busca padrão a ser aplicado nas bases de dados:

(novelty OR originality OR similarity) AND (“color palette” OR colors) AND (“user interface” OR gui OR “visual design”)

As *strings* de busca, conforme a formatação de cada repositório, são apresentadas na Tabela 5. Para o Google Scholar foi definida uma *string* especial para atender as limitações das opções de busca.

Repositório	***Search string***
ACM Digital Library	[[Abstract: novelty] OR [Abstract: originality] OR [Abstract: similarity]] AND [[Abstract: “color palette”] OR [Abstract: colors]] AND [[Abstract: “user interface”] OR [Abstract: gui] OR [Abstract:”visual design”]] AND [Publication Date: (01/01/2012 TO 18/04/2022)]

IEEE Xplore Digital Library	(((novelty OR originality OR similarity) AND (“color palette” OR colors OR “color set” OR “color space”) AND (interface OR design OR gui))) Filters Applied: 2012 - 2022
Scopus	TITLE-ABS (((novelty OR originality OR similarity) AND ("color palette" OR colors) AND ("user interface" OR gui OR "visual design"))) AND (LIMIT-TO (PUBSTAGE , "final")) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (DOCTYPE , "ar")) AND (LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018) OR LIMIT-TO (PUBYEAR , 2017) OR LIMIT-TO (PUBYEAR , 2016) OR LIMIT-TO (PUBYEAR , 2015) OR LIMIT-TO (PUBYEAR , 2014) OR LIMIT-TO (PUBYEAR , 2013) OR LIMIT-TO (PUBYEAR , 2012))
Google Scholar	(Busca 1) originality "color palette" “user interface” GUI - Filters 2012-2022 (Busca 2) originality color “user interface” GUI - Filters 2012-2022 (Busca 3) similarity color “user interface” GUI - Filters 2012-2022

Tabela 5. Strings de busca

3.2. Execução da Busca

A busca foi realizada em Junho de 2022 pelo autor do trabalho e revisada pela orientadora e pela coorientadora do trabalho. A busca inicial resultou em 603 artigos (Tabela 6).

Fonte	Número de resultados de busca	Número de resultados analisados	Número de documentos potencialmente relevantes	Número de documentos relevantes
ACM	13	13	2	2
IEEE	14	14	2	0
Scopus	14	14	1	1
Google Scholar - Busca 1	162	162	4	0
Google Scholar - Busca 2	12.900	200	8	2
Google Scholar - Busca 3	16.900	200	15	1
Total (sem documentos duplicados)				**3**

Tabela 6. Número de artigos identificados por repositório e por fase de seleção

A partir do resultado inicial das buscas, foram selecionados artigos potencialmente relevantes de acordo com os critérios de inclusão e exclusão por meio de uma análise do título, resumo e palavras-chave de cada artigo, buscando confirmar a relevância dos trabalhos de acordo com os critérios de inclusão e exclusão. Dessa forma, foram analisados todos os artigos obtidos pelas buscas, exceto nas buscas que obtiveram mais de 200 resultados, em que apenas os primeiros 200 artigos encontrados foram analisados.

Em seguida, foram analisados os artigos potencialmente relevantes pela leitura do artigo na íntegra. Como resultado final foram identificados 3 artigos relevantes.

3.3. Extração de dados

Os dados foram extraídos dos artigos de forma a responder às perguntas de análise conforme especificado na Tabela 7.

Pergunta de análise	Item	Descrição
AQ1. Quais abordagens existem para a avaliação da originalidade da paleta de cores de aplicativos e quais suas características?	Referência	Indicando a referência do artigo
	Nome	Indicando o nome do artigo
	Breve descrição	Indicando uma breve descrição da abordagem
	Tipo de interface	Android, IOS, App Inventor, Desktop
	Contexto	Profissional (ex. clonagem de apps), Educacional
AQ2. Qual modelo de cores é utilizado para avaliar a originalidade da paleta de cores?	Referência	Indicando a referência do artigo
	Modelo de cor utilizado	RGB, HSV, CIE
AQ3. Quais técnicas de similaridade são adotadas na análise?	Referência	Indicando a referência do artigo
	Extração das cores	Indicando a forma que os dados sobre a paleta de cores são extraídos do software
	Análise de similaridade	Indicando qual metodologia foi usada pela abordagem para executar a análise
	Universo de referência	Indicando quais artefatos são usados para a comparação e quantos
AQ4. Como a qualidade das abordagens foi avaliada e quais as principais descobertas?	Referência	Indicando a referência do artigo
	Fator(es) de qualidade avaliado(s)	Indicando qual o(s) fator(es) de qualidade usado(s) para analisar a abordagem (precisão, correlação com avaliações humanas)
	Tamanho da amostra	Indicando qual o tamanho total da amostra da abordagem
	Métodos de análise - junto dos fatores ou separado	Indicando qual método de análise foi utilizado para avaliar a abordagem
	Resultados	Indicando quais foram os resultados obtidos pela abordagem

Tabela 7. Especificação dos dados extraídos

3.4. Análise das abordagens relevantes

Quais abordagens existem para a avaliação da originalidade/similaridade das cores e quais suas características?

Foram encontrados apenas 3 artigos (Tabela 8) que utilizam abordagens de comparação da similaridade/originalidade das cores usadas em diferentes interfaces.

Referência	Nome	Breve descrição	Tipo de interface	Contexto
(Araújo et al., 2017)	Colors Similarity Computation for User Interface Adaptation	O artigo propõe um algoritmo para calcular a similaridade entre cores que será utilizada para definir qual técnica de adaptação será utilizada para aquele tipo de daltonismo.	Web	Profissional

(Goree et al., 2021)	Investigating the Homogenization of Web Design: A Mixed-Methods Approach	Por meio de técnicas de visão computacional, é perceptível a homogeneização do design de interfaces web e este artigo discute suas causas, incluindo tópicos como a padronização do esquema de cores.	Web	Profissional
(Khoker et al., 2016)	A fast and effective image retrieval scheme using color-, texture-, and shape-based histograms	É proposto um esquema de recuperação de imagens baseado em conteúdo, por meio dos recursos de cor, textura e forma.	Desktop	Profissional

Tabela 8. Visão geral das abordagens

Dentre as abordagens relevantes analisadas, é possível observar que nenhuma está atrelada ao contexto educacional, sendo todas contextualizadas no ambiente profissional. Outra observação importante nesse primeiro momento é em relação às interfaces utilizadas, em que nenhum dos três artigos apresenta uma solução para aplicativos Android e, portanto, não há ocorrências envolvendo o uso do App Inventor.

Qual modelo de cores é utilizado para avaliar a originalidade da paleta de cores?

Como parte da análise das informações sobre as abordagens é fundamental levar em consideração qual sistema de cores é utilizado em cada situação (Tabela 9).

Referência	**Modelo de cor utilizado**
(Araújo et al., 2017)	RGB
(Goree et al., 2021)	CIE L*c*h
(Khoker et al., 2016)	HSV (quantificado)

Tabela 9. Modelo de cor utilizado

Os artigos possuem distinções muito claras entre suas propostas, e uma característica discrepante entre as três propostas é o uso de modelos de cores variadas. Um modelo utiliza o sistema RGB, outro utiliza o CIE L*c*h (uma variante do CIE L*a*b) e outro utiliza o HSV. Dessa forma observa-se que parece não ter um consenso sobre o modelo de cores a ser utilizado neste tipo de contexto.

Quais técnicas de similaridade são adotadas na análise?

Para identificar como a similaridade das cores é calculada em cada abordagem, foram extraídas informações sobre a extração das cores, a análise da similaridade e o universo de referência (Tabela 10).

Referência	Extração das cores	Análise de Similaridade	Universo de referência
(Araújo et al., 2017)	Conjunto de cores pré-identificado	Propõe um algoritmo de similaridade de cores considerando o tipo de daltonismo do usuário como pesos no cálculo da distância das cores.	Comparação do um conjunto de cores de uma interface com o conjunto de cores predefinido do usuário.
(Goree et al., 2021)	As cores de uma interface são extraídas em um histograma	Apresenta duas formas distintas de analisar a similaridade: 1) Utilizando redes neurais convolucionais - CNN (*Convolutional Neural Networks*). Cada saída é um valor entre 0 e 1 que indica a similaridade estimada daquela página. Estas saídas são transformadas em um vetor que é usado para quantificar a diferença entre as imagens do site por meio do cálculo da distância Euclidiana. 2) Utiliza o EMD (*Earth Mover's Distance*) para medir a distância entre as cores em um histograma. O EMD captura a quantidade de "esforço" necessário para converter o esquema de cores de uma interface para outra. Nesse caso a similaridade é calculada utilizando a métrica CIE 1976 Delta E.	Comparação usando um conjunto de aproximadamente 200.000 *snapshots* em torno de 10.000 interfaces web.
(Khoker et al., 2016)	Para extrair os recursos de cores, é usado um histograma de cores no espaço de cores HSV quantizado	Ao comparar quatro medidas de similaridade (distância euclidiana, distância de Manhattan, Qui-quadrado e distância d_1) é decidido utilizar a métrica de distância d_1, visto que essa demonstrou o melhor resultado em termos dos testes de precisão e *recall*.	Conjuntos de dados de imagem de uso geral.

Tabela 10. Técnicas de similaridade nas abordagens

Quanto às técnicas de análise adotadas pelas abordagens é notável que também não há consenso. É possível observar que algumas métricas estão diretamente associadas ao sistema de cores utilizado na abordagem, como é o caso de Goree et al. (2021) que utiliza a equação de CIE 1976 Delta E para o cálculo da similaridade no sistema CIE L*c*h. Enquanto há outras que utilizam métricas desenvolvidas especialmente para a situação que deseja analisar, como em Araújo et al. (2017) que utiliza um algoritmo próprio para calcular a similaridade das cores considerando o tipo de daltonismo do usuário.

Vale mencionar que as escolhas das métricas são embasadas pela comparação de várias métricas. O caso mais explícito disso é apresentado pelo Khoker et al. (2016), que comparam as medidas de distância Euclidiana, a Distância de Manhattan, o Teste de χ^2 e a distância d1 (Agarwal et al., 2012). Sendo essa última escolhida por obter maior eficácia comparado com as demais.

Como uma abordagem alternativa, Goree et al. (2021) apresentam uma análise de capturas de tela de *websites* usando um modelo de *deep learning* de redes neurais convolucionais - CNN (*Convolutional Neural Networks*), que se baseia no treinamento de uma rede CNN para a identificar à qual empresa uma página web pertence, dada uma imagem dessa página como referência.

Como a qualidade das abordagens foi avaliada e quais as principais descobertas?

Após a abordagem ser desenvolvida, é importante avaliar a sua qualidade, chegando à uma conclusão por meio dos dados obtidos.

Referência	Fator(es) de qualidade avaliado(s)	Tamanho da amostra	Métodos de análise	Resultados
(Araújo et al., 2017)	Eficácia	Dois usuários	Qualitativo	Os resultados indicam que o algoritmo escolhe adequadamente a técnica de adaptação a ser utilizada para um determinado usuário, porém sem maiores conclusões especificamente sobre a análise da similaridade.
(Goree et al., 2021)	Sem informação	Sem informação	Sem informação	Sem informação
(Khoker et al., 2016)	Eficácia: Precisão e Revocação (*recall*)	Dois conjuntos contendo 1.000 imagens em um e 3.000 imagens no outro	Quantitativo	O uso dos histogramas normalizados fornece um conjunto robusto de recursos e garante que o sistema produza resultados altamente relevantes para o conteúdo da imagem na consulta. Mais especificamente, a parte de cores foi a que demonstrou maior eficácia, isto é, os melhores valores da relação entre precisão e revocação (*recall*).

Tabela 11. Avaliação da qualidade das abordagens

Porém, nem todas as abordagens apresentaram avaliações sobre o modelo demonstrado. Uma exceção é apresentado por Khoker et al. (2016), dos quais utilizaram testes com tamanhos de amostras relativamente grandes para avaliar a precisão e a revocação (*recall*) do modelo apresentado, obtendo um resultado satisfatório para a comparação das cores usando os histogramas normalizados e a métrica de distância d_1 para a similaridade das cores. Considerando que a métrica de distância d_1 possui o melhor valor médio de precisão (0,7559) e o melhor valor médio de revocação (0,1512) comparada às outras métricas.

3.5. Discussão

De forma geral, observa-se que atualmente as abordagens voltadas à avaliação da originalidade de paletas de cores de apps são quase inexistentes. Foram encontradas

somente três abordagens voltadas aos sistemas de software de forma geral, porém não foi encontrado nenhum especificamente voltado a apps, menos ainda a apps desenvolvidos com App Inventor. Observa-se também que mesmo entre as três abordagens não há muito consenso em relação aos modelos de cores a serem utilizados, e também não há consenso sobre o cálculo da similaridade a ser utilizado. Por exemplo, se for escolhido o sistema de cores CIE L*a*b poderia ser utilizada a métrica CIE 1976 Delta E como em Goree et al. (2021), no entanto, talvez poderia optar por uma versão mais atualizada desta métrica, como a CIE 2000 Delta E.

É possível notar que não foi encontrada nenhuma abordagem voltada ao contexto educacional, já que as abordagens relevantes estão inseridas apenas no contexto profissional. Além da falta de avaliações mais rigorosas da qualidade das abordagens propostas.

Assim, é perceptível que apesar das abordagens possuírem aspectos interessantes e úteis, nenhuma das três apresenta um modelo pronto que poderia ser utilizado no foco do presente trabalho. Dessa forma, observa-se que atualmente não existe ainda nenhuma abordagem que apresenta um modelo de avaliação da originalidade da paleta de cores no design de interface de apps Android desenvolvidos com App Inventor.

Ameaças à validade. Ao executar a revisão bibliográfica há o risco de omitir estudos relevantes. Visando mitigá-lo, foi definida uma *string* de busca para incluir todos os termos relevantes e seus sinônimos.

Para minimizar ainda mais as chances de omitir algum estudo relevante, a *string* foi utilizada para realizar buscas em diferentes bases de dados científicas (ACM, IEEE, Scopus e Google Scholar).

Quanto à mitigação dos riscos da seleção dos estudos e extração dos dados, foram definidos critérios de inclusão e exclusão detalhados que foram contemplados durante a busca. Além disso, o protocolo de seleção definido e executado pelo autor deste trabalho geraram resultados que foram revisados pela orientadora e coorientadora deste trabalho.

4. PROPOSTA DA SOLUÇÃO

Levando em consideração o estado da arte, a proposta deste trabalho busca fomentar o conhecimento sobre a originalidade de cores em aplicativos, principalmente no contexto educacional. Isso é feito ao desenvolver um modelo que avalia a originalidade da paleta de cores de um aplicativo feito com App Inventor em comparação com um universo de referência composto por outros aplicativos também feitos com App Inventor em contextos educacionais semelhantes.

4.1. Análise dos requisitos

Universo de referência: Para avaliar a paleta de cores é utilizado um conjunto de 1.773 diretórios, sendo que cada diretório contém as capturas de todas as telas de um determinado aplicativo (as imagens das capturas de tela estão no formato “png” com tamanho igual a 1.080 pixels de largura por 1.920 pixels de altura), totalizando 9.050 telas. O universo utilizado foi previamente obtido pela coleta das capturas de tela feita por membros do Grupo de Qualidade de Software (GQS).

Entrada: O usuário deve inserir o caminho de origem de uma pasta contendo capturas de todas as telas do aplicativo que deseja ter a originalidade de sua paleta de cores avaliada. As capturas de tela devem estar devidamente padronizadas seguindo os mesmos padrões utilizados para o universo de referência.

Saída: A abordagem desenvolvida visa avaliar o grau da originalidade referente a paleta de cores de apps feitos com App Inventor. Para isso é definida uma escala do grau de originalidade da paleta de cores que varia de 0 (paleta menos original) até 10 (paleta mais original) ao comparar a paleta com o universo de referência.

Analisando o contexto, são definidos os seguintes requisitos funcionais conforme apresentado na Tabela 12.

Requisito	Descrição	Artefato de entrada	Artefato de saída
Extração da paleta de cores	Por meio das capturas de telas de um determinado aplicativo, a solução extrai suas cores e mescla de acordo com a relevância, gerando uma paleta de cores proporcional do aplicativo	Capturas de tela (*Screenshots*) do aplicativo	Paleta de cores proporcional do aplicativo
Cálculo do grau da originalidade da paleta de cores de um aplicativo feito com App Inventor	O modelo é capaz de atribuir um valor numérico que determine o grau de originalidade da paleta de cores no design de interface de aplicativos desenvolvidos pelo App Inventor em relação a um universo de referência em conformidade com julgamentos humanos	Paleta de cores proporcional do aplicativo	Um valor numérico indicando o grau da originalidade da paleta de cores

Tabela 12. Requisitos da solução

4.2. Modelagem

A abordagem consiste em extrair as cores das capturas de telas do aplicativo, filtrar as cores mais relevantes gerando a paleta de cores proporcionais do aplicativo, comparar essa paleta com as paletas do universo de referência, atribuir um valor de originalidade em relação com cada paleta, para enfim avaliar o grau de originalidade do aplicativo (Figura 24).

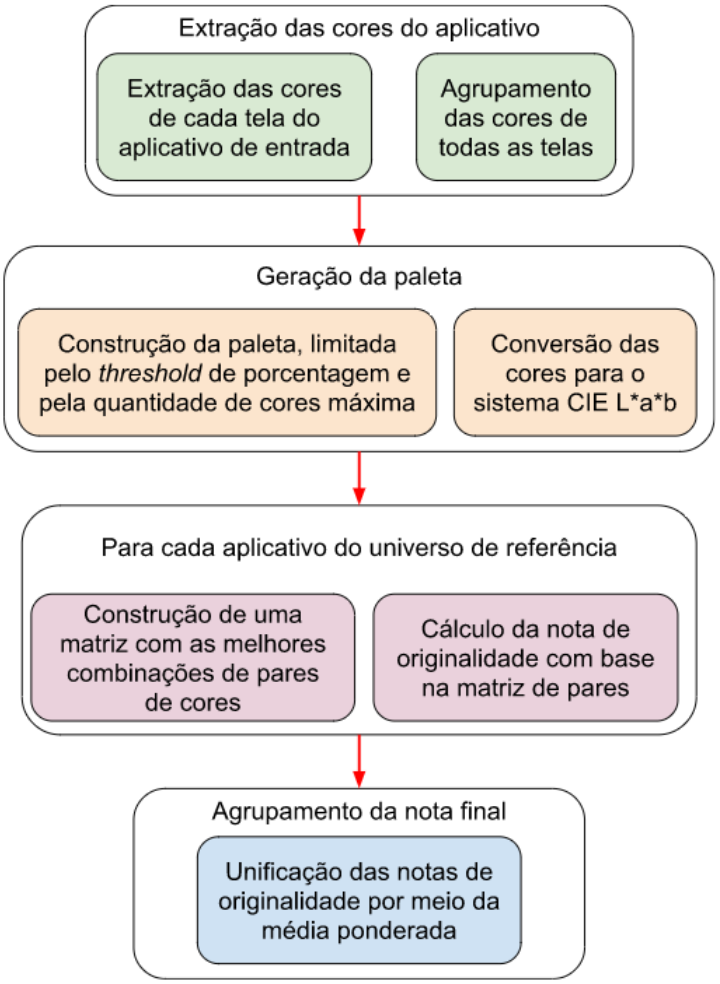


Figura 24. Ordem de execução das partes da modelagem

Como parte da solução proposta, a configuração da entrada esperada é dada por um conjunto de capturas de todas as telas do aplicativo que deve ser avaliado. Cada captura de tela deve estar no formato de imagem “png” com a resolução 1.080 pixels de largura por 1.920 pixels de altura para a orientação da página de retrato (Figura 25).

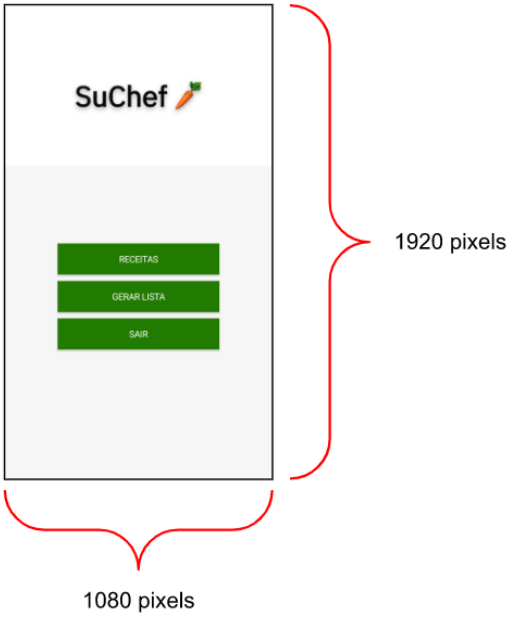


Figura 25. Exemplo de captura de tela no formato esperado

Extração das cores de um app. A extração das cores do aplicativo é feita por meio das capturas de tela geradas pelo aplicativo. Para extrair as cores é utilizada a biblioteca "*extcolors*" (Pypi, 2020) da linguagem Python que extrai as cores a partir de uma imagem. Por meio das cores extraídas de um aplicativo procura-se definir uma paleta de cores.

Identificação da paleta de cores de um app. Buscando definir, desenvolver e testar a percepção humana das cores da interface de diferentes aplicativos foi realizado um estudo, apresentado na próxima subseção, para auxiliar na escolha da forma de representação das cores de um aplicativo.

4.2.1 Estudo comparativo da originalidade das cores entre aplicativos

Com o objetivo de ter uma base para definir o cálculo das notas da solução desenvolvida é relevante produzir um estudo que considere a percepção humana real (questionando pessoas sobre a originalidade da paleta de cores em relação a um universo de referência pequeno). Além disso, o estudo aplicado busca responder duas perguntas:

P1: É possível usar uma paleta como representação das cores de um aplicativo para analisar sua originalidade cromática?

P2: Se for possível, é melhor utilizar a paleta em blocos (sem representação do espaço que as cores ocupam nas telas) ou a paleta proporcional (com

representação do espaço que as cores ocupam nas telas) para representar as cores de um aplicativo?

Como esse estudo é feito para avaliar a percepção humana, inicialmente estima-se (com base em uma comparação empírica) que um limite de no máximo 20 cores por paleta seja um valor razoável (quantidade de cores suficientes para julgar a originalidade da paleta, sem sobrecarregar o avaliador) para uma comparação visual entre as paletas. Sendo que estas são as cores que mais aparecem pelo aplicativo, resultado da soma das quantidades de pixels de todas as telas. Considerando que as cores presentes na paleta aparecem em pelo menos 1% do total de pixels do aplicativo (soma de todos os pixels de todas as telas).

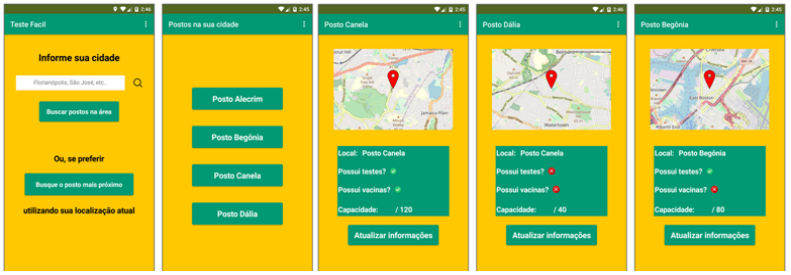
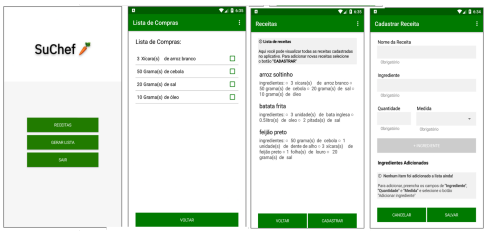
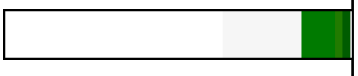
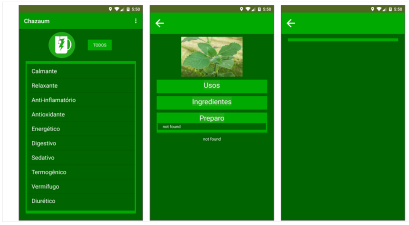
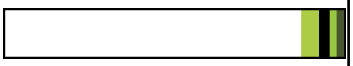
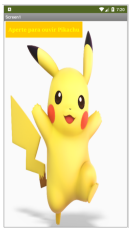
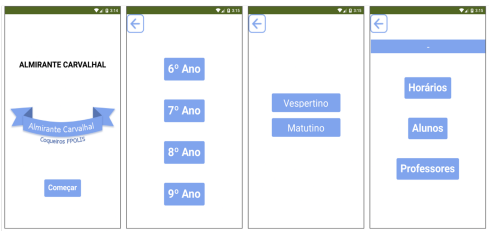
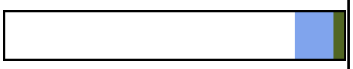
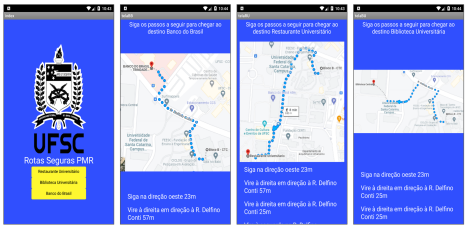
Screenshots	
Paleta em blocos	
Paleta proporcional	

Tabela 13. Capturas de tela e paletas de cores do aplicativo "TesteFacil"

O estudo foi realizado por meio da comparação de 10 aplicativos distintos (Tabela 14) com o aplicativo de exemplo "TesteFácil" (Tabela 13). A avaliação por cada participante é realizada usando classificações para avaliar do mais similar (nota 1) ao menos similar (nota 10).

App	Capturas de tela	Paleta em blocos	Paleta proporcional
suChef			

App	Capturas de tela	Paleta em blocos	Paleta proporcional
Chazaum			
Medike			
QArvore			
Pikachu			
AcheiOseuEmprego			
InfoCarvalhal			
RotasSeguras			

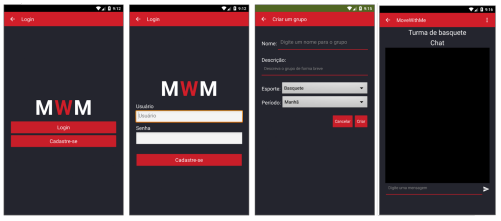
App	Capturas de tela	Paleta em blocos	Paleta proporcional
IMC			
Move With Me			

Tabela 14. Capturas de tela e paletas de cores dos demais aplicativos

O estudo foi realizado por meio de uma pesquisa no período de março a junho de 2022. Ao final foram contabilizadas as respostas de um total de 96 participantes convidados via redes sociais e emails, selecionados por conveniência. Sendo que a maioria (60 participantes) não possui experiência com design visual e/ou design de interfaces, 59 participantes estão na faixa etária de 18 a 29 anos e 6 responderam que possuem algum grau de daltonismo.

Considerando a avaliação das capturas de tela como a avaliação mais fiel à percepção humana para avaliar a similaridade entre os aplicativos (padrão ouro), é possível extrair as médias obtidas para cada aplicativo em cada cenário e compará-las entre si (Tabela 15).

App	Média das originalidades		
	Capturas de tela	Paleta em blocos	Paleta proporcional
suChef	3.2	4.6	4.6
Chazaum	3.3	4.7	4.6
Medike	3.7	4.6	3.2
QArvore	4.3	4.8	4.9
Pikachu	4.9	3.0	2.6
AcheiOseuEmprego	6.1	5.1	5.7
InfoCarvalhal	6.5	4.6	5.4
RotasSeguras	6.9	6.4	6.9
IMC	7.5	8.3	8.1
MoveWithMe	8.6	8.9	9.1

Tabela 15. Comparação entre as médias de similaridade de cada aplicativo em comparação ao aplicativo base “TesteFacil”

A interpretação extraída desses valores é de que quanto maior o valor da média obtida, menor o grau de originalidade entre a paleta de cores do aplicativo “TesteFacil” em comparação com outro aplicativo.

Pode-se observar inicialmente que ambas representações das paletas de cores possuem resultados similares com os obtidos diretamente das capturas de tela (Gráfico 1). Isso leva à conclusão de que é possível utilizar a representação de paleta de cores para expressar as cores de um aplicativo.

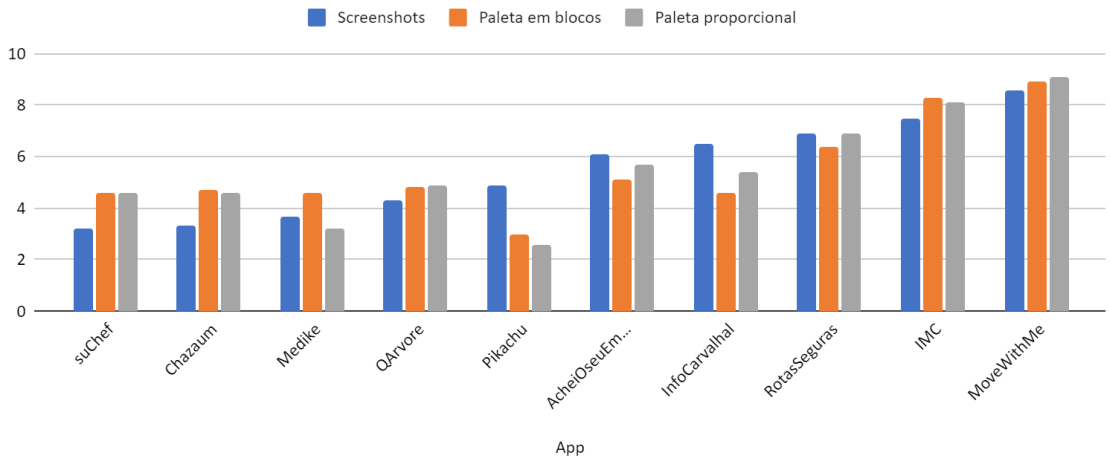


Gráfico 1. Comparação entre as médias de originalidade para capturas de tela (screenshots), paleta em blocos e paleta proporcional

Para confirmar essa semelhança entre as paletas e as capturas de tela e indicar qual seria a melhor forma de representação de paleta de cores, é possível utilizar a correlação de Spearman, analisando a correlação entre as duas representações de paletas individualmente com a representação direta utilizando somente as capturas de tela (Tabela 16). O coeficiente de correlação de Spearman é utilizado para indicar o quão próximos os valores de duas variáveis permanecem ao modificá-los, considerando que quanto mais próximo de um dos valores extremos (-1,1), maior a correlação entre as variáveis.

Correlação entre	ρ (rho)	valor-*p*
Capturas de tela e Paleta em blocos	0,6933037	0,0262
Capturas de tela e Paleta proporcional	**0,8267515**	0,00318

Tabela 16. Comparação entre os coeficientes de correlação de Spearman e os valores-p das paletas com a captura de tela

Considerando esses resultados, a representação utilizando a paleta proporcional possui maior correlação com as capturas de tela (ρ = 0,8267515), do que a paleta em blocos (ρ = 0,6933037).

O valor-*p* é utilizado para indicar a probabilidade de encontrar resultados que implicam que a correlação entre as capturas de tela e cada representação por paleta de cores é falsa. Portanto, os valores-*p* ínfimos da paleta em blocos (valor-*p* = 0,0262) e da paleta proporcional (valor-*p* = 0,00318) indicam que é muito provável que a representação em paletas de cores seja recomendada.

Sendo assim, pode-se concluir que a representação de paleta de cores proporcional se aproxima mais da percepção humana da paleta de cores em relação às interfaces de apps.

4.2.2 Definição do modelo de cores

Ao analisar os sistemas de cores utilizados nas pesquisas levantadas no estado da arte é possível obter a interpretação de que não há um consenso definido sobre qual seria o melhor sistema de cores para representar a paleta de cores de um aplicativo. Considerando isso, o sistema de cores escolhido para fazer isso é o CIE LAB (Ibraheem et al., 2012). A opção por esse sistema se deve ao fato de ser capaz de se assemelhar muito com a forma que o ser humano capta as cores pela visão, sendo possível julgar mais apropriadamente o nível de similaridade entre as cores (Ibraheem et al., 2012).

O universo de referência utilizado para comparar com a paleta de cores do aplicativo de entrada consiste em um conjunto de 1.773 aplicativos com no mínimo uma captura de tela em cada, totalizando 9.050 telas capturadas por pesquisadores da iniciativa Computação na Escola/INCod/INE/UFSC de apps da Galeria do App Inventor e desenvolvidos na iniciativa Computação na Escola. A captura de telas é realizada seguindo um roteiro para sistematicamente capturar todas as telas no mesmo formato (Lima, 2020).

4.3 Definição dos parâmetros da paleta de cores

Ao inserir as capturas de tela de um aplicativo como entrada da solução, é extraído o dicionário de frequência de cores RGB já buscando agrupar as cores similares (usando a medida “deltaE76” e o parâmetro de tolerância).

Em seguida, as cores que permanecem na paleta do aplicativo aparecem em uma porcentagem maior que um limite definido (chamado de *threshold* de porcentagem) do

total de pixels do aplicativo. Sendo que o total de cores que aparecem na paleta definitiva do aplicativo é limitada por um parâmetro denominado “quantidade de cores”.

Vale mencionar que a extração e geração das paletas de cores dos aplicativos do universo de referência é feita previamente utilizando as mesmas premissas.

Com as paletas de cores definidas é dado início ao processo de cálculo da nota de originalidade. Começando pela criação da matriz de similaridade entre dois aplicativos (aplicativo de entrada e aplicativo selecionado dentro do universo de referência) contendo todos os valores de similaridade de cada par de cores possíveis.

Após a geração da matriz, esta é utilizada para encontrar o conjunto das melhores combinações possíveis, construindo uma lista de pares de cores. Cada par é usado para fazer um cálculo individual que compõe a nota completa da originalidade para aquele app.

Esse processo é repetido até obter uma nota para cada aplicativo do universo de referência e, após um cálculo envolvendo a média simples e a média ponderada, é gerada a nota final da originalidade da paleta de cores do aplicativo de entrada.

4.3.1 Definição do grau de tolerância

Ao extrair as cores de um aplicativo é possível perceber que o total de cores obtidas (Tabela 17) é muito maior do que a percepção humana consegue compreender. Isso é notável principalmente quando há imagens ou efeitos de degradê no aplicativo.

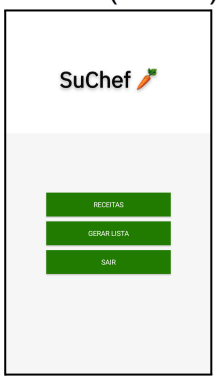
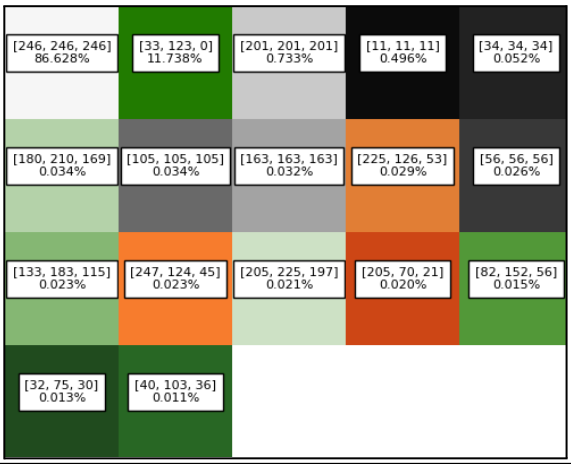
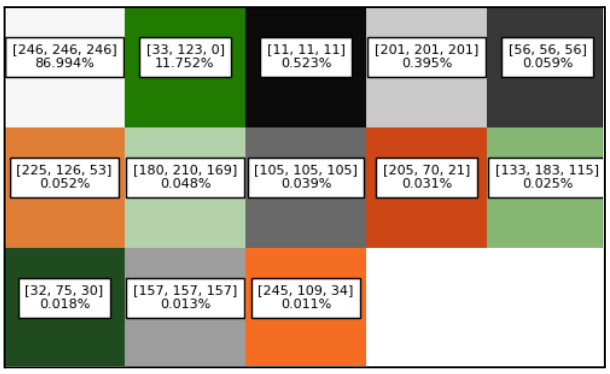
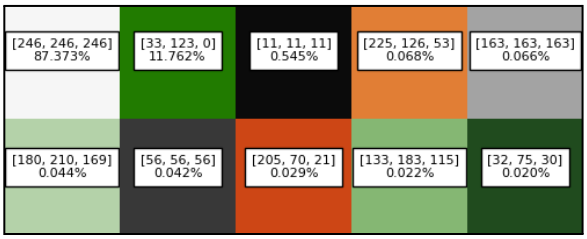
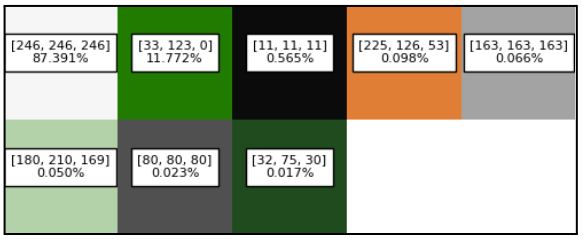
suChef (4ª tela)	Tolerância 10 - 1,1576 segundos	Tolerância 15 - 1,0889 segundos
		
	Tolerância 20 - 1,0881 segundos	**Tolerância 25 - 1,0834 segundos**
		

Tabela 17. Paletas de cores da 4ª tela do aplicativo “suChef” com diferentes tolerâncias

Para resolver esse problema de excesso de cores, é necessário mesclar cores muito semelhantes com o objetivo de reduzir o número de cores obtidas na extração (Tabela 18). A justificativa para essa mesclagem ser possível surge da impossibilidade de humanos perceberem tons muito próximos. Segundo Schuessler (2020), cores muito similares (valor da similaridade menor que 10) tendem a ser imperceptíveis pela visão humana.

Aplicativos (número da tela)	Total de cores extraídas
suChef (4ª tela)	4506
Meteorito (1ª tela)	57308
IMC (1ª tela)	394219
RotasSeguras (2ª tela)	18701
Medike (6ª tela)	2384
QArvore (1ª tela)	7091
vaiDeBike (4ª tela)	753
FloripaPraias (7ª tela)	16208
sos (1ª tela)	5949
Pikachu (1ª tela)	76599
WeLearn (1ª tela)	2523

Tabela 18. Quantidade de cores extraídas em telas de 11 aplicativos

Para realizar essa mesclagem das cores é utilizado o parâmetro de tolerância contido na função de extração "*extract_from_path()*" da biblioteca "*extcolors*" (Pypi, 2020). É por meio dele que se limita o quão similares as cores devem ser, para que ambas sejam unidas em apenas uma cor.

Ao comparar diferentes valores de tolerância para uma amostra de 11 telas variadas (Anexo A) é possível perceber que valores maiores de tolerância tendem a reduzir drasticamente o total de cores extraídas.

A biblioteca utilizada define como padrão o valor 32 para a tolerância. No entanto, ao incluir valores inferiores à 32, mas ainda elevados (como 20 e 25), é notável que há tons de cores possíveis de serem diferenciados pela percepção humana que são mesclados em uma única cor. Isso ocorre, por exemplo, na 6ª tela do aplicativo Medike (Tabela 19), em que as duas cores com maiores porcentagens são mescladas e, portanto, o tom mais escuro não aparece na paleta final usando tolerância 25, 20 ou até mesmo 15.

Valores inferiores a 10 não são considerados por ainda gerarem quantidades de cores elevadas, o que prejudicaria o tempo de execução no cálculo da originalidade posteriormente.

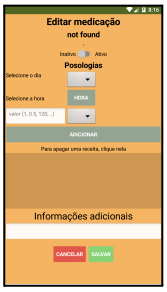
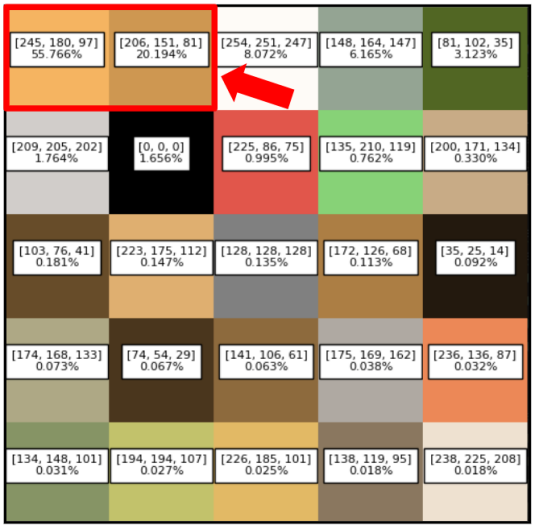
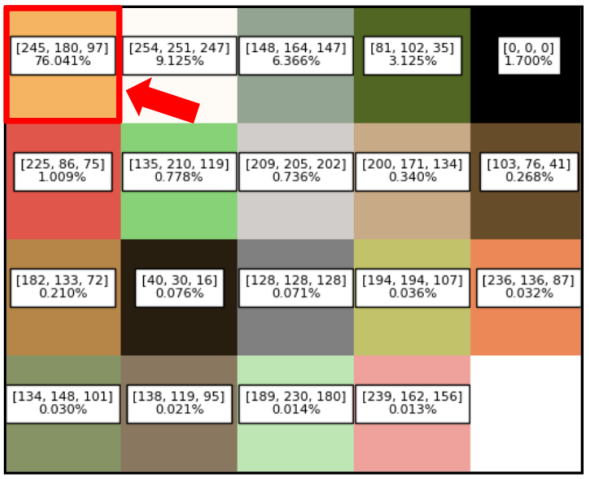
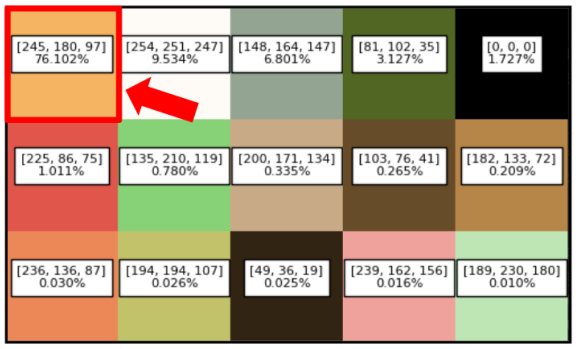
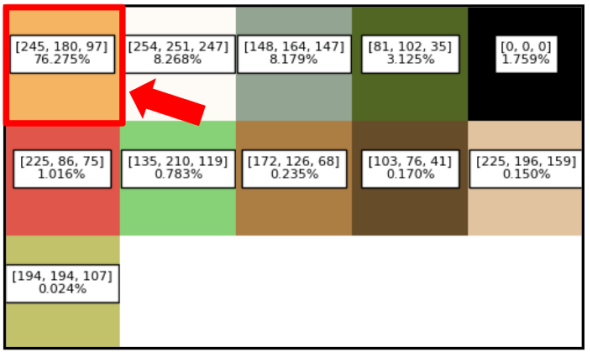
	Tolerância 10 - 1,0159 segundos	Tolerância 15 - 1,0463 segundos
Medike (6ª tela) 		
	Tolerância 20 - 1,0355 segundos	**Tolerância 25 - 1,0425 segundos**
		

Tabela 19. Paletas de cores da 6ª tela do aplicativo "Medike" com diferentes tolerâncias

Outra maneira de visualizar essa diferença entre as cores é utilizando as paletas de cores proporcionais (Anexo B) que deixam ainda mais explícito em que momento cada mesclagem ocorre. Por meio do estudo da percepção humana é possível constatar que a representação da paleta de cores de forma proporcional é uma maneira válida de representação e, portanto, será considerada na implementação da solução para avaliar a diferença na porcentagem das cores presentes nos aplicativos.

A Tabela 20 demonstra visualmente a importância do uso da tolerância para a mesclagem das cores pela queda drástica na quantidade de cores obtidas após a extração.

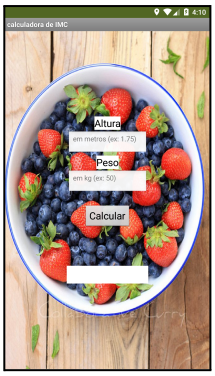
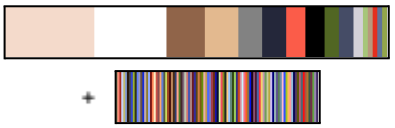
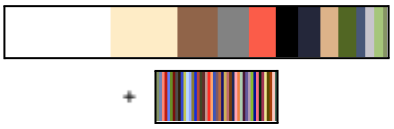
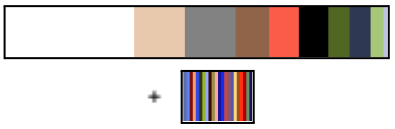
IMC (1ª tela)	Tolerância 10 (327 cores)	Tolerância 15 (125 cores)
		
	Tolerância 20 (57 cores)	**Tolerância 25 (32 cores)**
		

Tabela 20. Paletas proporcionais da tela do aplicativo "IMC" com diferentes tolerâncias

Entretanto, há telas, como a 4ª tela do aplicativo "suChef", que apresenta uma diferença muito pequena entre um tom de cinza e o branco que ainda é perceptível ao olho humano, o que faz com que ambas sejam mescladas em uma única cor até mesmo usando tolerância 10 (Tabela 21). Para situações como essa, outro fator deve ser levado em consideração, o tempo para a extração das cores da tela.

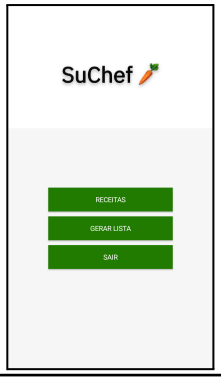
suChef (4ª tela)	Tolerância 10 (63 cores)	Tolerância 15 (30 cores)
		
	Tolerância 20 (21 cores)	**Tolerância 25 (14 cores)**
		

Tabela 21. Paletas proporcionais da tela do aplicativo "suChef" com diferentes tolerâncias

A diferença entre os dois tons da tela do aplicativo suChef somente é captada usando um valor de tolerância igual a 3 ou inferior. Contudo, utilizar um valor como esse para telas com quantidades de cores maiores tende a levar muito mais tempo (Tabela 22) para realizar a extração, como é o caso da 1ª tela do aplicativo IMC, que leva mais de 9 minutos para executar usando o valor de tolerância igual a 3.

App	Tolerância 0*	Tolerância 3	Tolerância 5	Tolerância 10	Tolerância 15	Tolerância 20	Tolerância 25
suChef	1,0894	1,5498	1,3571	1,1576	1,0889	1,0881	1,0834
RotasSeguras	1,2017	3,3788	2,1933	1,5653	1,3033	1,2616	1,2296
Meteorito	1,4626	12,1588	5,231	2,5544	1,8885	1,7593	1,6244

IMC	4,1117	561,7038	155,4983	29,9575	13,7138	9,7474	6,6271
Medike	0,9932	1,0839	1,0643	1,0159	1,0463	1,0355	1,0425
QÁrvore	1,1104	1,5211	1,364	1,12	1,1129	1,081	1,0626
WeLearn	0,9693	1,038	0,9888	1,0133	1,0362	1,0394	0,5024
vaiDeBike	0,9786	0,9536	0,9356	0,9958	1,0152	1,0234	1,0085
FloripaPraias	1,0991	4,3502	2,431	1,5147	1,3324	1,2659	1,1805
Pikachu	1,6715	22,6305	9,055	3,3529	2,5846	2,2949	1,8425
sos	0,4935	0,7547	0,6053	0,5446	0,5374	0,5471	0,5024

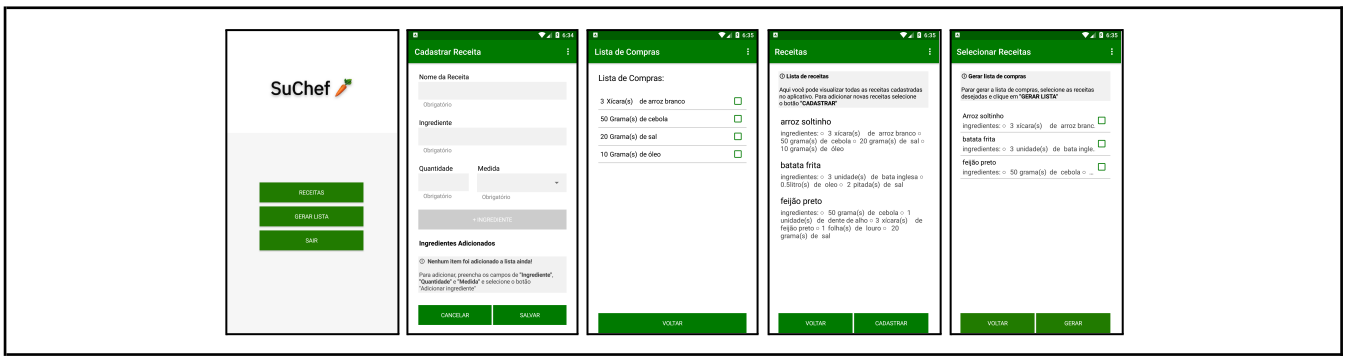
Tabela 22. Tempo de extração das cores das telas usando diferentes tolerâncias

*o valor de tolerância não faz nenhum agrupamento durante a extração

Portanto, para definir o valor de tolerância é necessário encontrar um equilíbrio entre o tempo de extração e o nível tolerável de mesclagem das cores. Consequentemente, o valor que melhor atende estes requisitos é o 10, pois mescla cores suficientemente próximas em um tempo razoável considerando situações extremas.

4.3.2 Definição do valor limite de *threshold* de porcentagem

O *threshold* de porcentagem (Anexo C) é um parâmetro utilizado para delimitar a porcentagem mínima que uma cor deve apresentar para ser incluída na paleta de cores do aplicativo. Esse valor é definido principalmente pela relevância das cores selecionadas, tomando o cuidado para não deixar o valor tão baixo ao ponto de ser quase irrelevante. Conforme apresentado pelo exemplo da Tabela 23, é perceptível que usar os valores 1,5%, 1% e 0,5% reduzem consideravelmente a quantidade de cores que devem aparecer na paleta final.



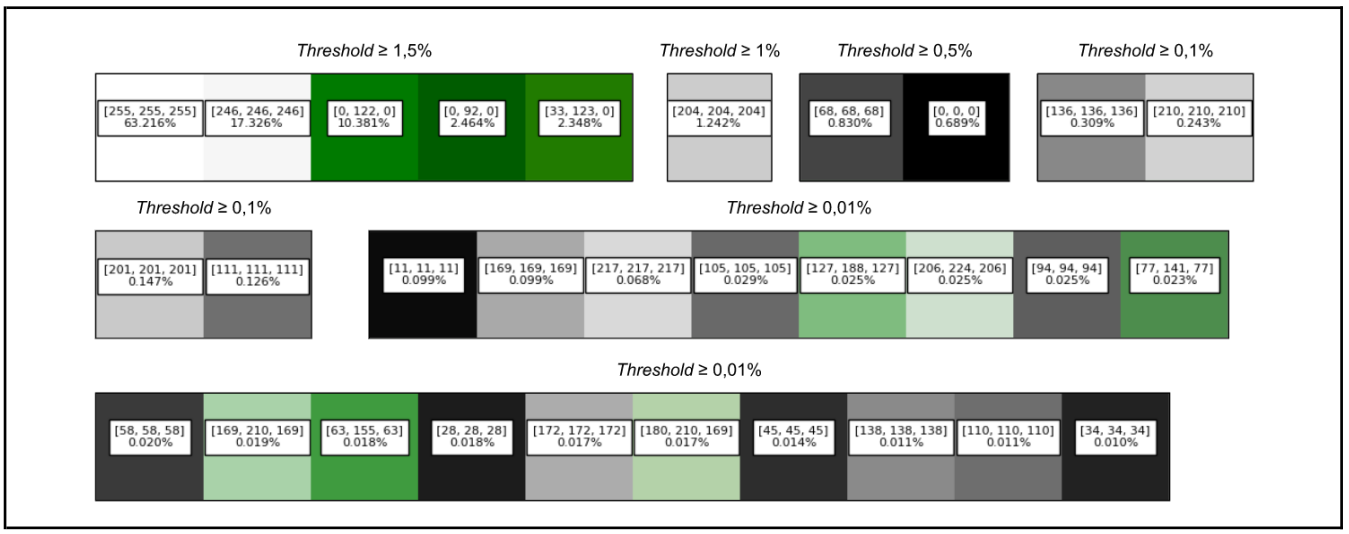


Tabela 23. Paleta de cores do aplicativo "suChef", usando diferentes valores de threshold

Contudo, a perda por usar esses valores pode ser grande em algumas situações. Como é o caso tanto da paleta do aplicativo "Medike" (Tabela 24) quanto do aplicativo "vaiDeBike" (Tabela 25), em que ambos possuem botões das cores verde e vermelho que somente são incluídos na paleta se for considerado um *threshold* de 0,1% ou inferior.

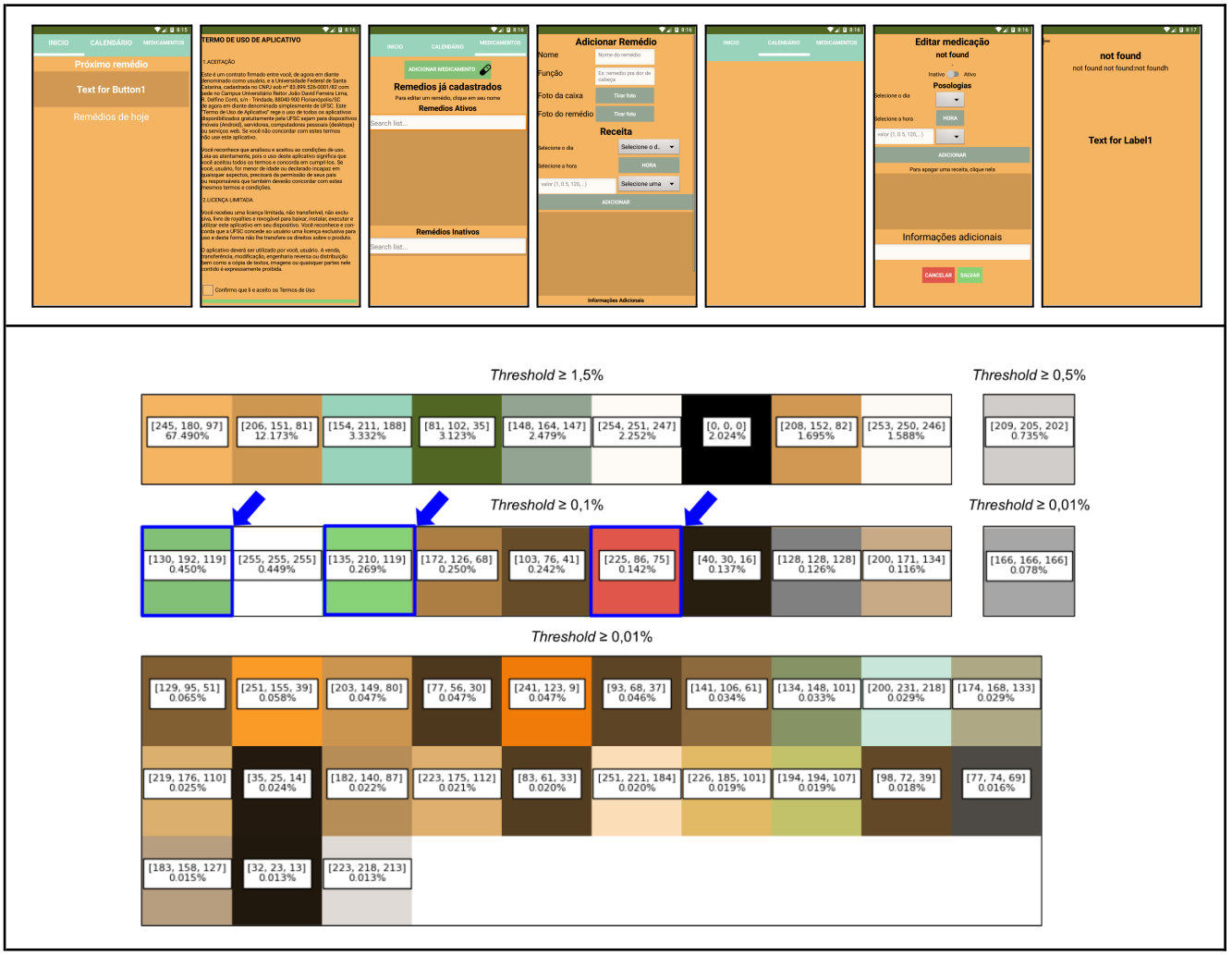


Tabela 24. Paleta de cores do aplicativo "Medike" com diferentes valores de threshold

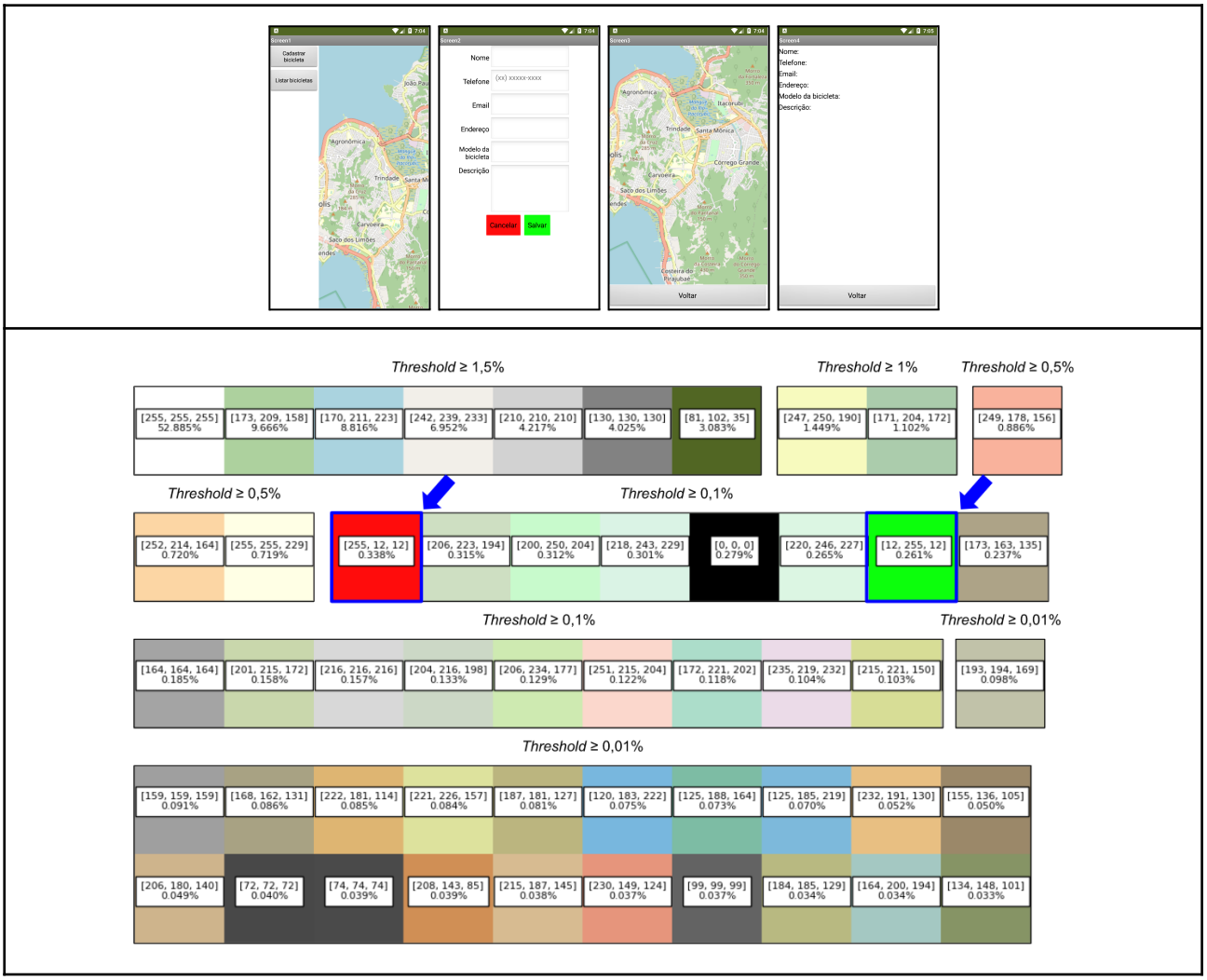


Tabela 25. Paleta de cores do aplicativo "vaiDeBike" com diferentes valores de threshold

Há determinados casos em que somente um *threshold* de 0,01% seria capaz de captar algumas cores perceptíveis pela visão humana. Contudo, permitir essa flexibilidade faz com que cores demais sejam incluídas na paleta (Gráfico 2), tornando o parâmetro inviável e os cálculos de originalidade mais custosos. Por isso, o valor escolhido foi 0,1% por ser o melhor equilíbrio entre essas situações.

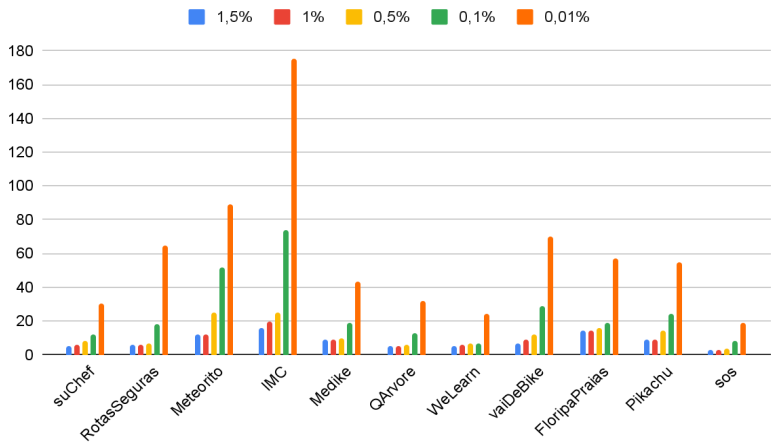


Gráfico 2. Quantidades de cores dos aplicativos com diferentes valores de threshold

4.3.3 Definição da quantidade máxima de cores na paleta

O parâmetro que limita a quantidade de cores máxima que uma paleta pode possuir, é fundamental para o cálculo da similaridade, pois por meio dele é possível reduzir o tempo de execução.

O ponto de partida para definir a quantidade máxima de cores na paleta final vem da quantidade total de cores presentes nas paletas extraídas usando os valores 10 para a tolerância e 0,1% para o *threshold* da porcentagem, pois esse total é usado como base para definir o limite máximo de cores que uma paleta pode conter.

Ao gerar um gráfico das quantidades de cores resultantes da extração feita (Gráfico 3), é possível perceber que, com exceção dos aplicativos “Meteorito” e “IMC” (que possuem muitas cores por conterem ícones e imagens), nenhum outro aplicativo ultrapassa a marca de 30 cores totais. Indicando que utilizar um limite superior a 30 cores pode ser ineficaz.

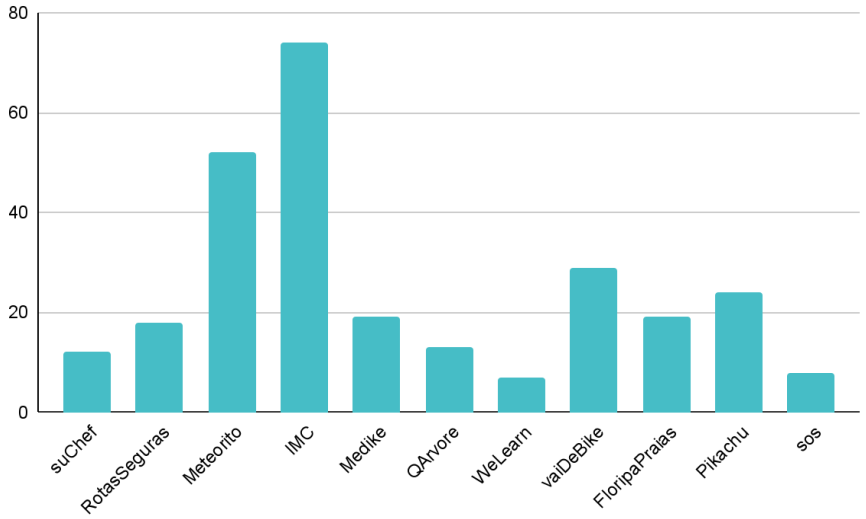


Gráfico 3. Quantidades totais de cores para apps com tolerância 10 e threshold 0,1%

Outra métrica importante que auxilia na decisão de qual é o limite da quantidade de cores utilizada para calcular a similaridade a ser definido, é o tempo de execução para diferentes valores de quantidade. Os resultados obtidos (Tabela 26) ao testar em um cenário extremo, em que ambos aplicativos comparados possuem o máximo de cores, apresentam um aumento considerável no tempo, quando mais cores são incluídas no cálculo. Vale considerar que esses valores são referentes à comparação entre apenas dois aplicativos. Portanto, ao comparar com um universo de referência, esse tempo deve ser multiplicado pelo total de aplicativos contidos no universo de referência, o que torna as diferenças de tempo ainda mais distantes.

Quantidade de cores	Tempo (segundos)
5	0,0393
10	0,1464
15	0,3351
20	0,5919
25	0,9333
30	1,3091
35	1,8638
40	2,4098
45	3,0407
50	3,8937

Tabela 26. Tempos para calcular a originalidade usando diferentes quantidades de cores

Considerando o tempo de execução é possível concluir que a melhor opção seria utilizar um valor limite da quantidade de cores muito pequeno, reduzindo drasticamente o tempo total de execução da solução. Contudo, como visto anteriormente pelo Gráfico 3 (quantidades totais de cores), usar um valor de limite muito pequeno descartaria uma quantidade de cores significativas da paleta do aplicativo. Por isso, o valor limite da quantidade de cores escolhido é 20, pois o tempo de execução pode ser tolerável (dependendo da quantidade de aplicativos contidos no universo de referência) sem cortar cores relevantes na maioria dos casos.

Com isso, fica definido que a paleta de cores definitiva de um aplicativo deve ser extraída usando valor de tolerância 10, valor limite de *threshold* de porcentagem igual a 0,1% e a quantidade de cores presentes na paleta limitada a 20 cores no máximo.

Aplicando esses parâmetros para os aplicativos de exemplo, são obtidas as paletas de cores apresentadas na Tabela 27.

suChef	RotasSeguras	Meteorito	IMC	Medike	QArvore
					
WeLearn	vaiDeBike	FloripaPraias	Pikachu	sos	
					

Tabela 27. Paletas de cores resultantes dos 11 aplicativos exemplares

4.4 Cálculo da originalidade entre dois aplicativos

O cálculo da originalidade entre dois aplicativos é usado para atribuir uma nota de originalidade da paleta de cores ao aplicativo dado como entrada em relação a outro selecionado.

O universo de referência utilizado para demonstração inclui todas as telas dos mesmos 10 aplicativos usados no estudo comparativo da originalidade da paleta de cores de um aplicativo pela percepção humana. Uma paleta é gerada para cada um dos aplicativos e fica armazenada em um arquivo externo do formato “json”, contendo as informações da cor (valor em CIE L*a*b) e sua porcentagem de presença em relação ao total de pixels do aplicativo.

Em termos conceituais, a paleta do aplicativo dado como entrada é comparada individualmente com as paletas de cada aplicativo contido no universo de referência. Ao selecionar um aplicativo do universo de referência, as cores de ambos aplicativos são “pareadas”, isto é, cada cor do aplicativo de entrada é colocada em um par com a respectiva cor mais similar do aplicativo selecionado. Assim, a nota da originalidade da paleta de cores é resultado do somatório de cálculos de similaridade feitos para cada par de cores gerados.

Inicialmente, uma função denominada “*app_pair_palette_similarity_matrix”* recebe duas paletas de dois aplicativos distintos (aplicativo de entrada e um aplicativo selecionado do universo de referência) e avalia qual das duas possui uma menor quantidade de cores em sua composição, para ajustar ambas com a mesma quantidade. Assim, uma matriz de similaridade é gerada (Figura 26) de forma que as linhas representam as cores de uma paleta, as colunas representam as cores da outra paleta e as células são preenchidas com o valor de similaridade (calculado pela função “*delta_E*” da biblioteca “*colour*”) das duas cores contidas naquele índice.

```
lines, columns = len(real_input_app), len(real_universe_app)
originality_matrix = [[0 for x in range(lines)] for y in range(columns)]
i = 0
# Compara cada cor do aplicativo de entrada com cada cor do aplicativo do universo de referência atribuindo
# uma nota de originalidade entre eles
for color_input_app in real_input_app:
    j = 0
    for color_universe_app in real_universe_app:
        originality_matrix[i][j] = colour.delta_E(color_input_app[0], color_universe_app[0], method='CIE 2000')
        j += 1
    i += 1
return originality_matrix
```

Figura 26. Trecho do código da criação da matriz de originalidade dentro da função “app_pair_palette_similarity_matrix”

A matriz de similaridade é passada para uma outra função denominada "*optimized_map_sim_palette*", responsável por atribuir uma nota de originalidade da primeira paleta em comparação com a segunda.

Para melhor avaliar a similaridade entre as cores é necessária uma comparação par a par das cores de uma paleta para a outra. Isso é feito com o auxílio da função "*compute*" da biblioteca "*munkres*", que percorre pela matriz procurando a melhor combinação das cores de cada paleta. Nesse caso, a função é baseada no algoritmo húngaro (Kuhn, 2012), um algoritmo de otimização combinatorial que, tendo uma matriz como entrada, encontra as melhores combinações possíveis em tempo polinomial.

Em outras palavras, ao selecionar uma cor da primeira paleta, a função indica qual cor da segunda paleta possui a maior similaridade e une ambas em um par. Contudo, se a função encontra uma cor da primeira paleta ainda mais similar com a cor da segunda paleta atribuída a um par anteriormente, um novo par é formado com essas duas cores e a outra cor da primeira paleta, que havia ficado sem par, recebe a segunda cor mais similar com ela (Figura 27).

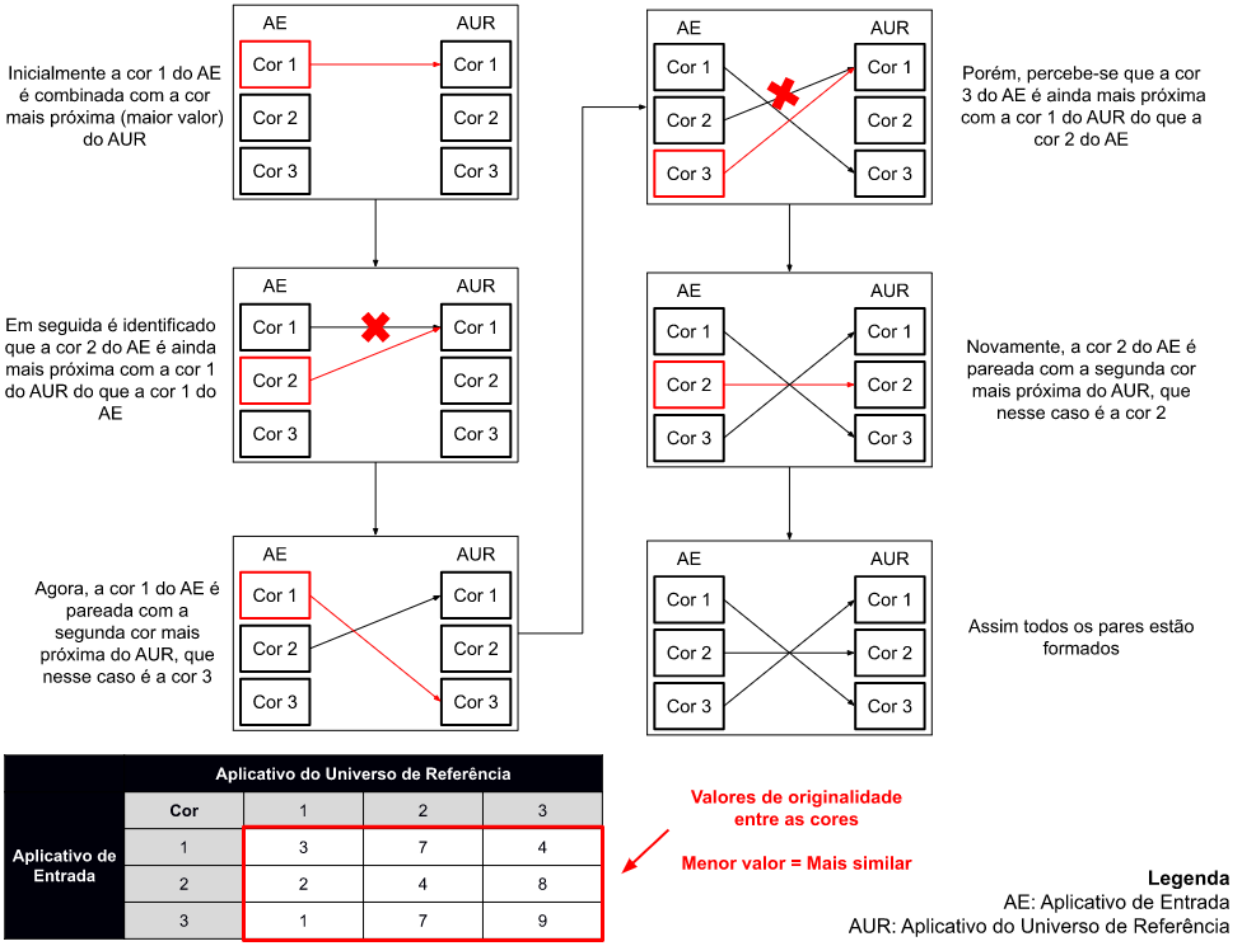


Figura 27. Passos da execução do algoritmo húngaro na função "compute"

Um exemplo da execução do algoritmo húngaro pode ser visto na Figura 28, em que algumas cores do aplicativo “TesteFacil” são pareadas com determinadas cores do aplicativo “Medike”.

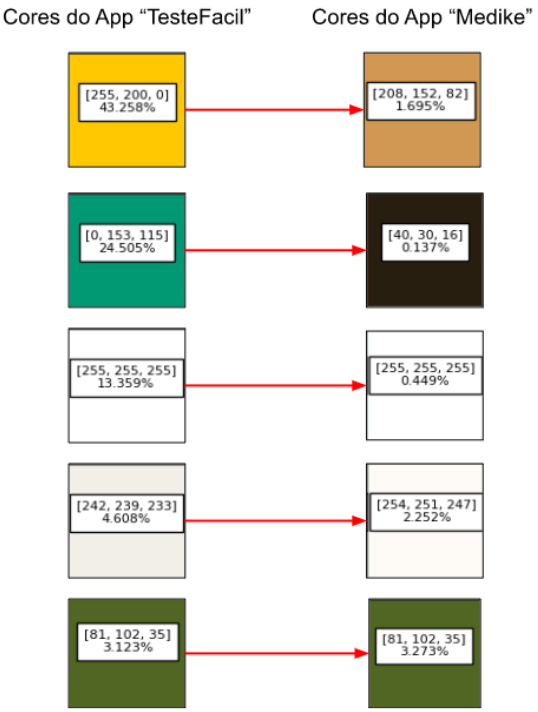


Figura 28. Pares das primeiras cores do app “TesteFacil” com as cores do app “Medike”

Com os pares formados, o cálculo da nota de originalidade da paleta pode ser efetivamente feito (Figura 29).

$$nota = \sum\{[(|pctg_1 - pctg_2| \times peso_{pctg} + \Delta E2000(cor_1, cor_2) \times peso_{\Delta E}] \times pctg_1 \times ajuste\}$$

Figura 29. Cálculo da nota de originalidade da paleta de cores de um aplicativo

A nota é resultado do somatório de “notas individuais” dadas para cada par de cores definidos anteriormente, sendo que cada uma dessas “notas” terá seu valor ajustado tanto pela porcentagem que a cor do aplicativo principal ocupa “$pctg_1$” e por um fator de “*ajuste”* que permite formatar a nota para o formato mais adequado (nesse caso o valor utilizado é 0,1 para que a nota esteja entre 0 e 10).

A “nota individual” de cada cor é gerada pela soma da diferença entre as porcentagens das duas cores e do valor de similaridade calculado pelo “*delta_E*” já contido na matriz. No entanto, esses dois valores precisam de um ajuste para dar uma nota mais realista ao julgamento humano.

Com a finalidade de obter os valores de “$peso_{pctg}$” e “$peso_{\Delta E}$” foram executados diversos testes combinando valores variados para ambos. O fator “$peso_{pctg}$” recebeu valores variando em 0,05 dentro do intervalo [0,4, 1], já o fator “$peso_{\Delta E}$” recebeu valores

variando em 0,05 dentro do intervalo [1, 3]. O motivo para o uso de valores maiores para o "$peso_{\Delta E}$" vem de ser o elemento principal dentro da equação.

O valor do peso foi determinado de forma empírica comparando as notas obtidas pela avaliação da originalidade da paleta do aplicativo "TesteFacil" com as paletas de cada aplicativo contido no estudo apresentada na seção 4.2.1, analisando a proximidade dos valores calculados com os valores obtidos pela avaliação humana na pesquisa.

De forma mais simplificada, calcula-se a nota de originalidade do aplicativo "TesteFacil" com cada um dos aplicativos do estudo de percepção humana para diferentes valores do "$peso_{pctg}$" e "$peso_{\Delta E}$". As notas são comparadas com as notas originais obtidas no estudo, de forma que a diferença entre as médias seja a menor possível, indicando assim a melhor combinação dentro dos intervalos sugeridos. Dessa maneira, a melhor combinação obtida é 0,4 para "$peso_{pctg}$" e 2,6 para "$peso_{\Delta E}$" com uma diferença média de aproximadamente 1,12 pontos para mais ou para menos de cada paleta em comparação com os valores obtidos na pesquisa (Tabela 28).

App	Nota pela percepção humana	Nota do cálculo da solução	Diferença entre as notas
suChef	4,59	6,31	1,72
Chazaum	4,59	4,1	0,49
Medike	3,22	5,2	1,98
QArvore	4,86	4,04	0,82
Pikachu	2,57	3,36	0,79
AcheiOseuEmprego	5,73	4,88	0,85
InfoCarvalhal	5,38	6,92	1,54
RotasSeguras	6,88	5,08	1,8
IMC	8,09	6,92	1,17
MoveWithMe	9,08	9,12	0,04
Média	5,5	4,11	1,12

Tabela 28. Comparação das notas de originalidade do aplicativo "TesteFacil" pela percepção humana e pelo cálculo da solução

Vale relembrar que os valores variam de 0 (menos original) até 10 (mais original). Como "$peso_{\Delta E}$" é 2,6, em uma situação em que duas paletas são suficientemente distintas, a nota ultrapassa o valor 10. Para contornar esse caso, uma condição é

adicionada para definir a nota como 10 se ela ultrapassar esse valor. Isso tende a ser útil principalmente para flexibilizar a nota dentro do contexto da Educação Básica.

4.5 Cálculo da nota final de originalidade

Com as notas de originalidade geradas a partir da comparação com cada paleta do universo de referência, é necessário utilizar algum meio de agrupá-las em uma única nota. A maneira mais eficaz encontrada de fazer esse agrupamento é por meio da média ponderada (Figura 30).

$$nota\ final = \frac{\sum(nota_{atual} \times peso_{atual})}{\sum(peso_{atual})}$$

Figura 30. Cálculo da nota final da originalidade da paleta do aplicativo de entrada

Para calcular essa média ponderada é preciso atribuir diferentes pesos para as notas obtidas. Isso é feito com a ajuda de uma média simples (Figura 31) de todas as notas (em que todas as notas possuem o mesmo peso).

$$média_{simples} = \frac{\sum nota_{atual}}{tamanho\ do\ universo\ de\ referência}$$

Figura 31. Cálculo da média simples de todas as notas

O cálculo do "$peso_{atual}$" (peso da nota atual do aplicativo do universo de referência comparado) é feito ao obter o valor absoluto da diferença entre a "$nota_{atual}$" com o valor da média simples (Figura 32). De forma conceitual, quanto mais distante uma nota estiver da média simples, maior será seu peso para o cálculo da nota final. A escolha do peso de ser calculado dessa forma vem da ideia de que notas mais distantes, sejam notas próximas a 10 ou próximas a 0, devem ter um impacto maior por representarem comparações entre paletas muito ou pouco originais entre si.

$$peso_{atual} = ceil(|média_{simples} - nota_{atual}|)$$

Figura 32. Cálculo do peso da nota do universo de referência selecionado

A função "*ceil*" é utilizada para arredondar o valor obtido da subtração para o próximo inteiro superior a esse valor, mantendo todos os pesos como inteiros maiores ou iguais a 1.

Seguindo a ideia de que notas mais distantes têm maior impacto na nota final, existem duas situações especiais em que o peso é ainda maior (acrescenta mais 1 ao "$peso_{atual}$"), notas menores que 2 ou maiores que 7 (Figura 33). Esses casos foram

definidos pois são as notas mais importantes contidas dentro de todo o conjunto. Inicialmente os valores deveriam ser equivalentes (2,5 para o limite inferior e 7,5 para o limite superior), porém essa proporção foi desequilibrada para flexibilizar um pouco mais as notas dentro do contexto da Educação Básica.

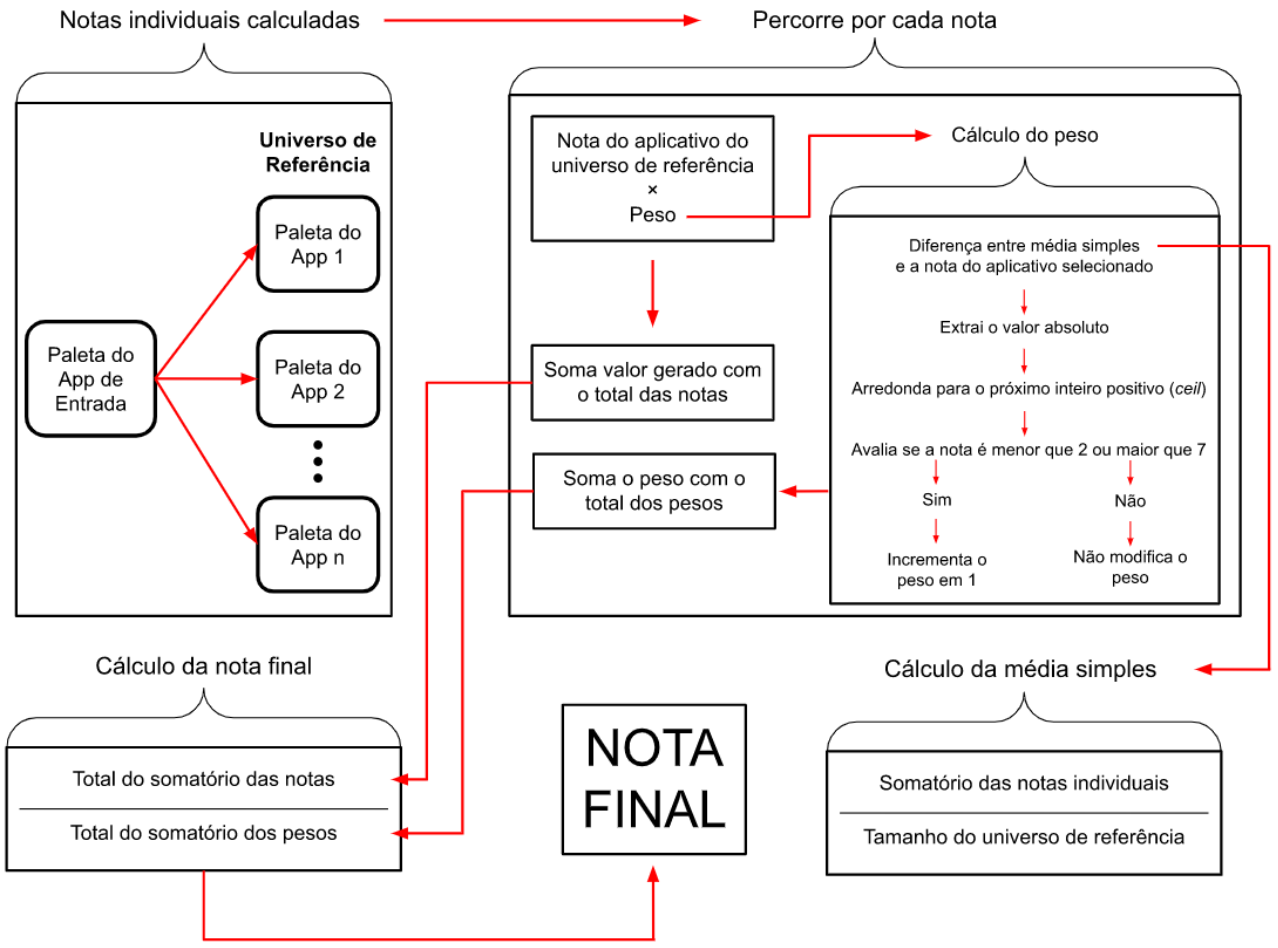


Figura 33. Representação do cálculo da nota final da originalidade

Um outro caso a ser levado em consideração é a situação de haver uma nota 0 dentro do conjunto de notas. Neste caso, a nota final será 0 independente do valor da média ponderada, porque para uma paleta atingir a nota 0 ela precisa ser exatamente idêntica a outra, o que leva a paleta a não ser original.

4.6 Avaliação do cálculo da nota final de originalidade

A média simples das notas de originalidade dos aplicativos utilizados na pesquisa da percepção humana em comparação com o aplicativo “TesteFacil” (Tabela 28) é 5,5. Aplicando o mesmo cálculo da média ponderada com as notas individuais obtidas na pesquisa e essa média simples resulta na nota 6.

Enquanto que o cálculo feito dentro da solução, comparando os mesmos aplicativos da pesquisa da percepção humana com o aplicativo “TesteFacil”, a nota obtida é 6,5, apenas 0,5 a mais que a nota original da pesquisa.

A diferença entre as notas utilizando a média ponderada é muito pequena. Ao comparar essa diferença com as notas obtidas pela média simples, é notável que utilizar a média ponderada é mais viável que a média simples. A diferença entre as médias simples do estudo da percepção humana e da nota obtida automaticamente pela solução é de quase 1,5. A média simples das notas da percepção humana é 5,5 e a média simples da solução automatizada é 4,11. Enquanto que a média ponderada apresenta uma diferença de apenas 0,5 entre as médias, quase 1 ponto a menos de diferença.

Sendo assim, com base na comparação observa-se que a solução proposta pode ser considerada precisa, pelo motivo da nota de originalidade obtida pelo aplicativo de exemplo se aproximar suficientemente da nota alocada pelas pessoas que responderam ao estudo da percepção humana.

Em termos do tempo de execução, o cálculo leva aproximadamente 1 minuto e 35 segundos para retornar a nota final de originalidade do aplicativo “TesteFacil” ao comparar com os 1773 aplicativos do universo de referência, sendo 19 segundos para gerar a paleta do aplicativo dado como entrada e 76 segundos para calcular a originalidade. Tempos inferiores poderiam ser obtidos ao reduzir o limite da quantidade de cores presentes nas paletas, porém isso pode influenciar negativamente a nota final por ter uma quantidade menor de cores sendo comparadas. Assim, por meio dos resultados do tempo total de execução para calcular o grau de originalidade (executando em uma máquina Linux, com processador Intel Core i5 e 32 gigabytes de memória RAM), assume-se que o tempo de aproximadamente 1 minuto e 35 segundos é considerado razoável, mesmo não sendo tão próximo de um tempo “instantâneo”, ideal para o contexto da Educação Básica. Na prática, é possível considerar que se trata de um tempo longo demais. Uma alternativa para aprimorar esse tempo seria calibrar melhor o universo de referência, observando o comportamento dos aplicativos em diferentes avaliações para descartar aqueles que não agregam nenhum valor para o cálculo da nota, reduzindo ele para um tamanho mínimo, porém ainda representativo.

4.7 Implementação

A implementação da solução é feita por meio da linguagem de programação “Python 3.10” (https://www.python.org/). Uma linguagem escolhida pela simplicidade e

familiaridade com a linguagem, além de haver bibliotecas importantes para o desenvolvimento.

A arquitetura da solução (Figura 34) é separada em 4 arquivos, sendo 3 classes e 1 arquivo contendo funções adicionais usadas pelas classes. Sendo que o código fonte da implementação está disponível em um repositório do *gitlab* (Anexo D).

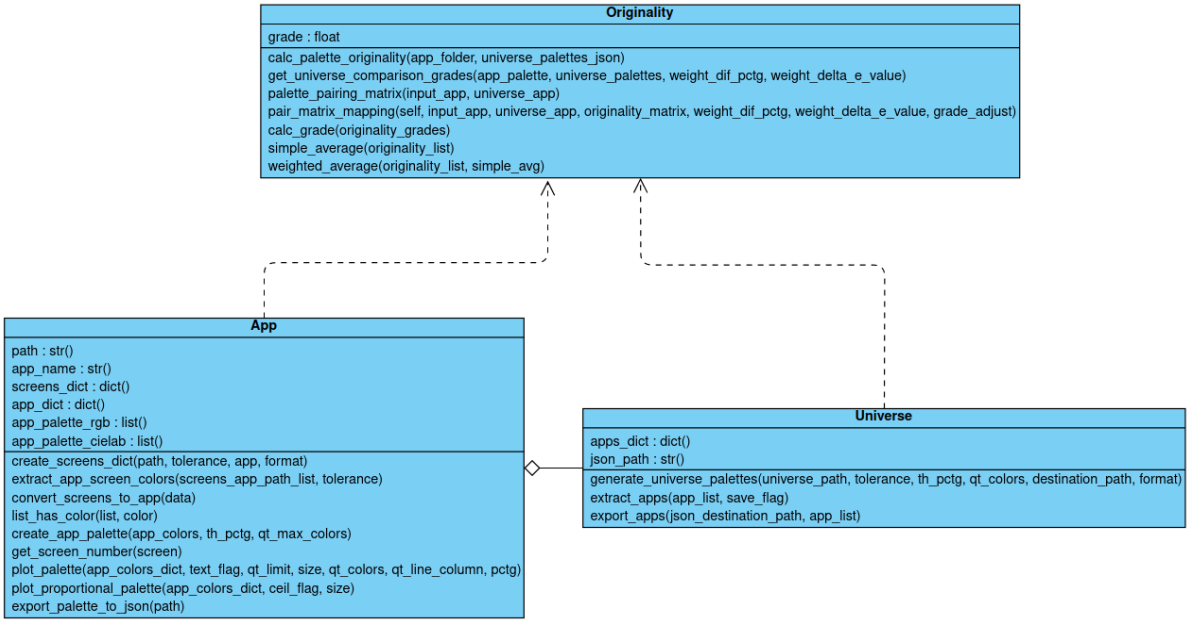


Figura 34. Diagrama de classes da implementação

4.7.1 Classe “App”

A classe inicial da solução é a classe “App”. Ela contém todas as funções que envolvem a extração, conversão e construção da paleta de cores de um aplicativo individualmente.

Ao chamar o construtor da classe (Figura 35), uma sequência de funções são acionadas para gerar a paleta de cores do aplicativo com os parâmetros (tolerância, *threshold* de porcentagem de cores e limite máximo da quantidade de cores) desejados.

```
# Método construtor da classe "App"
def __init__(self, path, tolerance, th_pctg, qt_colors, app_name=""):
    # O "path" é o caminho para o diretório onde estão as capturas das telas do aplicativo
    self.path = path
    # Recebe o nome do aplicativo e o dicionário de cores do aplicativo por telas
    self.app_name, self.screens_dict = self.create_screens_dict(path, tolerance, app_name)
    # As cores das telas do aplicativo são agrupadas em um só dicionário
    self.app_dict = self.convert_screens_to_app(self.screens_dict)
    # A paleta do aplicativo é extraída mantendo o sistema de cores original (RGB)
    self.app_palette_rgb = self.create_app_palette(self.app_dict[1], th_pctg/100, qt_colors)
    # A paleta de cores é convertida para o sistema de cores CIE L*a*b
    self.app_palette_cielab = self.convert_to_CIELAB_app(self.app_palette_rgb)
```

Figura 35. Método construtor da classe “App”

Começando pela função “*create_screens_dict*” (Figura 36), cujo objetivo é extrair as cores de cada tela do aplicativo, criando listas de cores associadas às suas respectivas quantidades de pixels e porcentagem de presença na tela. Essa função pode funcionar em dois modos diferentes. Se o parâmetro “*path*” recebe uma lista dos caminhos de cada uma das telas do aplicativo, então o algoritmo pode passar para a parte de extração. Já se o parâmetro receber um caminho no formato “*string*”, então é preciso obter os caminhos de cada tela do aplicativo antes da extração.

```python
# Função responsável por criar um dicionário contendo informações de quantidades e porcentagens
# das cores em cada uma das telas do aplicativo
def create_screens_dict(self, path, tolerance=10, app="", format="png"):
    # Valores iniciais da lista das telas e do nome do aplicativo
    # Se o parâmetro "path" contém a lista pronta dos caminhos das telas, esses valores permanecem assim
    screen_list = path
    app_name = app

    # Caso o parâmetro "path" contém um caminho para o diretório do aplicativo, então os caminhos das telas e
    # o nome do aplicativo são coletados
    if type(path) == str:
        screens_paths_list = get_all_app_screens_path(path, "linux", format)
        app_name = list(screens_paths_list.keys())[0]
        screen_list = screens_paths_list[app_name]

    # Aqui são efetivamente extraídas todas as cores de cada uma das telas e são adicionadas à suas respectivas
    # chaves dentro do dicionário de telas do aplicativo
    contScreens = 0
    screens_dict = dict()
    for screen in screen_list:
        contScreens += 1
        screen_colors_pixel_list, screen_colors_pctg_list, _ = self.extract_app_screen_colors([screen],
                                                                                               tolerance)
        text = self.get_screen_number(screen)
        screens_dict[text] = [screen_colors_pixel_list, screen_colors_pctg_list]
        print(f"App: {app_name} | Tela: {text}\nTotal de telas do app: {contScreens}")

    print(f"App {app_name} processado com sucesso!\n")
    return app_name, screens_dict
```

Figura 36. Função “create_screens_dict” da classe “App”

Dentro da função “*create_screens_dict*” (Figura 36), a função “*extract_app_screen_colors*” (Figura 37) é chamada para que a extração das cores, seja efetivamente feita por intermédio do método “*extract_from_path()*” da biblioteca “*extcolors*” (Pypi, 2020). A extração utiliza o valor de tolerância fornecido como parâmetro e a função retorna a lista da quantidade de pixels de cada cor, a lista da porcentagem dos pixels de cada cor e a quantidade total de pixels da tela.

```python
# Função de extração das cores de cada uma das telas de um aplicativo
def extract_app_screen_colors(self, screens_app_path_list, tolerance=10):
    screen_colors_pixel_list = [None] * len(screens_app_path_list)    # lista de cores por tela (por pixels)
    screen_colors_pctg_list = [None] * len(screens_app_path_list)     # lista de cores por tela (por porcentagem)
    screen_pixel_count = [None] * len(screens_app_path_list)          # lista de contagem de pixels por tela

    # Percorre por cada tela, extraindo as cores e mesclando de acordo com o valor de tolerância dado
    i = 0
    for screen in screens_app_path_list:
        screen_colors_pixel_list[i], screen_pixel_count[i] = extcolors.extract_from_path(screen, tolerance)
        screen_colors_pctg_list[i] = tup_list_to_sorted_dict(screen_colors_pixel_list[i], screen_pixel_count[i])
        i = i + 1
    return screen_colors_pixel_list, screen_colors_pctg_list, screen_pixel_count
```

Figura 37. Função "extract_app_screen_colors" da classe "App"

Em sequência, o construtor chama a função "*convert_screens_to_app*" (Figura 38). Essa função agrupa as listas de cores de cada tela em apenas uma lista de pixels e uma lista de porcentagens, obtendo assim uma representação unificada de todas as cores do aplicativo.

```python
# Função agrupa as listas de cores de cada tela do aplicativo em apenas uma
def convert_screens_to_app(self, data):
    pixel_count = 0
    colors_list = []
    # Percorre por cada tela do aplicativo, inserindo a quantidade de cada cor em uma lista única
    for screen_key in data.keys():
        for color in data[screen_key][0][0]:
            index = self.list_has_color(colors_list, color[0])
            # Se a cor já está presente na lista, sua quantidade é incrementada
            if index >= 0:
                colors_list[index][1] += color[1]
            # Se a cor não está presente na lista, é incluída
            else:
                colors_list.append(list(color))
            # Mantém uma contagem de todos os pixels acumulados até então
            pixel_count += color[1]

    # Calcula a porcentagem da presença de cada cor em relação ao total de pixels do aplicativo
    new_pctg_list = copy.deepcopy(colors_list)
    for i in range(0, len(new_pctg_list)):
        new_pctg_list[i][1] = new_pctg_list[i][1] / pixel_count

    return [colors_list, new_pctg_list]
```

Figura 38. Função "convert_screens_to_app" da classe "App"

A próxima etapa é dada pela função "*convert_to_CIELAB_app*" (Figura 39), que converte a lista de cores fornecida como parâmetro do sistema de cores RGB para o sistema de cores CIE L*a*b.

```python
# Função converte todas as cores da paleta para o sistema CIE L*a*b
def convert_to_CIELAB_app(self, data):
    new_data = copy.deepcopy(data)
    for color_index in range(0, len(data)):
        new_data[color_index][0] = rgb_to_CIELAB(data[color_index][0])

    return new_data
```

Figura 39. Função "convert_to_CIELAB_app" da classe "App"

Tendo uma lista com todas as cores do aplicativo e suas devidas porcentagens, basta chamar a função “*create_app_palette*” (Figura 40), responsável por construir a paleta definitiva do aplicativo, usando os valores do *threshold* de porcentagem e o limite máximo da quantidade de cores passados como parâmetro da função.

```
# Função define a paleta de cores do aplicativo utilizando os parâmetros de threshold de porcentagem da cor e
# de quantidade máxima de cores
def create_app_palette(self, app_colors, th_pctg, qt_max_color):
    palette = []
    for color in app_colors:
        # Se a porcentagem da cor for maior ou igual que o valor passado como parâmetro,
        # a cor deve ser considerada para uma possível inclusão na paleta
        if color[1] >= th_pctg:
            # Se a paleta estiver vazia já pode adicionar a cor
            if len(palette) == 0:
                palette.append(color)
            else:
                # Se o tamanho da paleta for menor que o limite máximo de cores passado como parâmetro,
                # o tamanho utilizado para verificar se a cor deve ser inserida é igual ao tamanho atual da paleta
                # caso contrário, o tamanho será o próprio valor passado como parâmetro
                if len(palette) < qt_max_color:
                    size = len(palette)
                else:
                    size = qt_max_color
                # Se o tamanho é menor que o valor passado como parâmetro e a porcentagem da última cor
                # presente atualmente na paleta é maior ou igual que a porcentagem da cor selecionada,
                # basta adicionar essa cor ao fim da paleta
                if size < qt_max_color and palette[size - 1][1] >= color[1]:
                    palette.append(color)
                else:
                    # Caso contrário, é necessário encontrar a posição ideal para inserir essa nova cor,
                    # mantendo a paleta ordenada pela porcentagem
                    for i in range(0, size):
                        if palette[i][1] < color[1]:
                            if size < qt_max_color and size == len(palette):
                                palette.append(palette[size - 1])
                            for j in range(len(palette) - 1, i, -1):
                                palette[j] = palette[j - 1]
                            palette[i] = color
                            break

    return palette
```

Figura 40. Função “create_app_palette” da classe “App”

4.7.2 Classe “Universe”

A classe “Universe” existe para implementar as funções relacionadas ao universo de referência. Logo em seu método construtor (Figura 41) é possível importar o dicionário de paletas dos aplicativos por meio de um arquivo do formato “*json*” ou executar uma função que gera as paletas de todos os aplicativos contidos em um diretório, a partir de um caminho fornecido.

```python
# Método construtor da classe "Universe"
def __init__(self, universe_path, tolerance=10, th_pctg=0.1, qt_colors=20, destination_path="", format="png",
             import_flag="False"):
    # Se a flag de importação estiver ativa, o dicionário de aplicativos é importado de um arquivo json
    if import_flag:
        self.apps_dict = import_json(universe_path)
        self.json_path = universe_path
    # Caso a flag de importação estiver inativa, o dicionário de aplicativos precisa
    # ser gerado a partir do caminho passado como parâmetro
    else:
        self.apps_dict = self.generate_universe_palettes(universe_path, tolerance, th_pctg, qt_colors,
                                                         destination_path, format)
        self.json_path = destination_path
```

Figura 41. Método construtor da classe "Universe"

A função "*generate_universe_palettes*" (Figura 42) instancia um objeto da classe "App" para cada aplicativo encontrado no diretório do universo de referência. Como o método construtor da classe "App" cria a paleta de cores do aplicativo automaticamente, é necessário somente chamar a função "*get_app_palette_cielab*" para obter a paleta e adicioná-la no dicionário das paletas de todos os aplicativos do universo de referência.

```python
# Função que extrai as paletas de todos os aplicativos contidos no caminho do universo de referência
def generate_universe_palettes(self, universe_path, tolerance, th_pctg, qt_colors, destination_path="",
                               format="png"):
    # Coleta os caminhos das telas de todos os aplicativos do universo de referência de uma só vez
    screens_paths_list = get_all_app_screens_path(universe_path, "linux", format)
    apps_dict = dict()
    # Para cada aplicativo contido no universo de referência, sua paleta é adicionada no dicionário de aplicativos
    for app in screens_paths_list:
        current_app = App(screens_paths_list[app], tolerance, th_pctg, qt_colors, app)
        apps_dict[app] = current_app.get_app_palette_cielab()
    if destination_path != "":
        export_json(destination_path, apps_dict)

    print(f"Universo de referência processado com sucesso!\n"
          f"Total de aplicativos processados: {len(apps_dict)}")
    return apps_dict
```

Figura 42. Função "generate_universe_palettes" da classe "Universe"

4.7.3 Classe "Originality"

Dentro da classe "Originality" estão contidas as funções responsáveis pelo cálculo da nota de originalidade da paleta de cores de um aplicativo. O construtor (Figura 43) chama apenas uma função que retorna a nota final recebida pela aplicação de entrada.

```python
# Método construtor da classe "Originality"
def __init__(self, app_path, universe_palettes_json):
    self.grade = self.calc_palette_originality(app_path, universe_palettes_json)
```

Figura 43. Método construtor da classe "Originality"

A função “*calc_palette_originality*” (Figura 44) executa uma sequência de funções que quando combinadas geram uma nota de originalidade da paleta. Nessa função é possível encontrar marcadores de tempo, importados pela biblioteca “*timeit*” (Python, 2022), para cronometrar o tempo de execução de cada etapa do código (extração e avaliação), a fim de avaliar se a solução executa em um tempo razoável.

Para gerar a paleta de cores do aplicativo de entrada, é instanciado um objeto da classe “App” usando os mesmos valores de tolerância, *threshold* de porcentagem e limite máximo da quantidade de cores, definidos na modelagem. De forma similar, as paletas do universo de referência são importadas ao instanciar um objeto da classe “Universe” e passar o caminho do arquivo “*json*” como parâmetro.

```
# Função que chama as funções de geração da paleta de cores do aplicativo de entrada, importação das paletas
# do universo de referência e de cálculo da originalidade
def calc_palette_originality(self, app_folder, universe_palettes_json):
    start_generate = timeit.default_timer()
    # Instancia a paleta de cores do aplicativo de entrada
    app = App(app_folder, 10, 0.1, 20)
    end_generate = timeit.default_timer()

    # Importa as paletas do universo de referência
    universe = Universe(universe_palettes_json, import_flag=True)
    universe_palettes = universe.get_apps_dict()
    start_calc = timeit.default_timer()

    # Gera as notas ao comparar com cada aplicativo do universo de referência e agrupa em uma única nota
    universe_grades = self.get_universe_comparison_grades(app.get_app_palette_cielab(), universe_palettes, 0.4, 2.6)
    grade = self.calc_grade(universe_grades)
    end_calc = timeit.default_timer()

    print("\nTempo para gerar a paleta do aplicativo:", round(end_generate - start_generate, 2))
    print("\nTempo para calcular a originalidade do aplicativo:", round(end_calc - start_calc, 2))
    print("A nota da originalidade da paleta de cores do aplicativo", app.get_app_name(), "é", grade)
    return grade
```

Figura 44. Função “calc_palette_originality” da classe “Originality”

Em seguida, a função “*get_universe_comparison_grades*” (Figura 45) é chamada. Essa função simplesmente percorre pelos aplicativos do universo de referência e inclui a nota recebida pela comparação de cada um com o aplicativo de entrada dentro de um dicionário de notas.

```
def get_universe_comparison_grades(self, app_palette, universe_palettes, weight_dif_pctg=0.55,
                                   weight_delta_e_value=1.5):
    app_sim_un_apps_dict = dict()
    # Para cada aplicativo do universo de referência, a matriz de originalidade e a nota são geradas
    for app in universe_palettes.keys():
        originality_matrix = self.palette_pairing_matrix(app_palette, universe_palettes[app])
        print("Ao comparar com o aplicativo", app)
        originality = self.pair_matrix_mapping(app_palette, universe_palettes[app], originality_matrix,
                                               weight_dif_pctg, weight_delta_e_value)
        app_sim_un_apps_dict[app] = originality
    return app_sim_un_apps_dict
```

Figura 45. Função “get_universe_comparison_grades” da classe “Originality”

A nota de originalidade ao comparar a paleta do aplicativo de entrada com a paleta de um aplicativo do universo de referência é gerada pela combinação de duas funções. A primeira delas, "*palette_pairing_matrix*" (Figura 46), constrói uma matriz que contém todos os valores de originalidade ao comparar cada cor do aplicativo de entrada com cada cor do aplicativo do universo de referência.

```
# Função que cria uma matriz contendo todos os valores de originalidade ao comparar as cores das paletas
# do aplicativo de entrada e de um aplicativo do universo de referência
def palette_pairing_matrix(self, input_app, universe_app):
    # Os dois aplicativos são ajustados para terem tamanhos iguais
    real_input_app = input_app
    real_universe_app = universe_app
    if len(input_app) > len(universe_app):
        real_input_app = input_app[:len(universe_app)]
    elif len(universe_app) > len(input_app):
        real_universe_app = universe_app[:len(input_app)]

    lines, columns = len(real_input_app), len(real_universe_app)
    originality_matrix = [[0 for x in range(lines)] for y in range(columns)]
    i = 0
    # Compara cada cor do aplicativo de entrada com cada cor do aplicativo do universo de referência atribuindo
    # uma nota de originalidade entre eles
    for color_input_app in real_input_app:
        j = 0
        for color_universe_app in real_universe_app:
            originality_matrix[i][j] = colour.delta_E(color_input_app[0], color_universe_app[0], method='CIE 2000')
            j += 1
        i += 1
    return originality_matrix
```

Figura 46. Função "palette_pairing_matrix" da classe "Originality"

A matriz de originalidade é então repassada como parâmetro para a função "*pair_matrix_mapping*" (Figura 47), que inicialmente computa a melhor combinação possível de pares de cores pelos valores de originalidade, ou seja, executa o algoritmo húngaro (Kuhn, 2012) implementado pela função "*compute*" da biblioteca "*Munkres*".

Ao obter os pares de cores, a função calcula uma nota de originalidade para cada par de cores, fazendo um somatório dessas cores para gerar uma nota geral da comparação com a paleta de cores do aplicativo do universo de referência. A fórmula utilizada aqui é a mesma apresentada na Figura 29, sendo que ambos parâmetros de peso recebem os valores 0,4 e 2,6 definidos previamente.

```python
# Função que encontra as melhores combinações possíveis de pares para os valores de originalidade contidos
# na matriz, calcula a nota de cada par de cores e retorna a nota geral da originalidade do aplicativo de entrada
# em comparação com o aplicativo do universo de referência
def pair_matrix_mapping(self, input_app, universe_app, originality_matrix, weight_dif_pctg=0.55,
                        weight_delta_e_value=1.5, grade_adjust=0.1):
    # Encontra as melhores combinações possíveis de cores, usando os valores da matriz de originalidade
    m = Munkres()
    indexes = m.compute(originality_matrix)

    # Para cada par de cores definido
    grade = 0
    for row, column in indexes:
        # Seleciona o valor de originalidade do par de cores atual
        delta_e_value = originality_matrix[row][column]
        # print(f'({row}, {column}) -> {delta_e_value:.2f}  \t||\t', end='')

        # Seleciona as porcentagens de cada uma das duas cores
        pctg1 = input_app[row][1]
        pctg2 = universe_app[column][1]

        # Cálculo da originalidade
        # O peso da diferença da porcentagem de cores interfere diretamente no resultado
        dif_pctg = abs(pctg1 - pctg2) * weight_dif_pctg
        # O peso da diferença do valor de originalidade (similaridade delta e) de cores
        # também interfere diretamente no resultado
        delta_e_value_weighted = delta_e_value * weight_delta_e_value
        # O ajuste de nota influencia na magnitude da nota
        calc = (dif_pctg + delta_e_value_weighted) * pctg1 * grade_adjust
        grade = grade + calc
    grade = round(grade, 2)
    if grade > 10:
        grade = 10

    print('A nota de originalidade é: %f' % grade)
    return grade
```

Figura 47. Função “pair_matrix_mapping” da classe “Originality”

Tendo o dicionário com as notas de comparação com cada aplicativo do universo de referência, é necessário unificar todas as notas. Essa unificação é iniciada pela função “*calc_grade*” (Figura 48) que obtém a média simples de todas as notas, calculada pela função “*simple_average*” (Figura 49) e fornece a mesma como parâmetro para a função “*weighted_average*” (Figura 50), retornando uma nota que varia em 0,5 dentro do intervalo de 0 (paleta nada original) até, no máximo, 10 (paleta muito original).

```python
# Função que chama o cálculo da nota pela média ponderada, usando a média simples como parâmetro e mantendo a nota
# como um múltiplo de 0.5
def calc_grade(self, originality_grades):
    avg = self.simple_average(originality_grades)
    if avg == 0:
        return 0
    return math.ceil(self.weighted_average(originality_grades, avg) * 2) / 2
```

Figura 48. Função “calc_grade” da classe “Originality”

```python
# Função que calcula a média simples de todas as notas obtidas
def simple_average(self, originality_list):
    avg = 0
    for app in originality_list.keys():
        if originality_list[app] == 0:
            return 0
        avg += originality_list[app]
    return avg / len(originality_list)
```

Figura 49. Função “simple_average” da classe “Originality”

A função “*weighted_average*” (Figura 50) segue as regras definidas na seção 4.3.5 para calcular a média ponderada de todas as notas obtidas. Sendo que quanto mais afastada da média simples uma nota está, maior será o peso e o impacto na nota final.

```
# Função que calcula a média ponderada de todas as notas obtidas, com pesos específicos para cada cenário
def weighted_average(self, originality_list, simple_avg):
    total_weight = 0
    sum = 0
    # Para cada nota obtida ao comparar o aplicativo de entrada com cada aplicativo do universo de referência
    for app in originality_list.keys():
        grade = originality_list[app]
        # O peso é resultado da diferença entre a nota de originalidade ao comparar com o app atual
        # e a média simples de todas as notas, arredondando o valor absoluto disso para o próximo inteiro positivo
        weight = math.ceil(abs(simple_avg - grade))
        # Se a nota é menor que dois, maior que 7 ou se o peso estiver definido como 0, o peso é incrementado em 1
        if grade <= 2 or grade >= 7 or weight == 0:
            weight += 1
        sum += grade * weight
        total_weight += weight
    return sum / total_weight
```

Figura 50. Função “weighted_average” da classe “Originality”

4.4.4 Funções adicionais

Para facilitar a solução, algumas funções adicionais foram incluídas em um arquivo separado. As principais funções contidas no arquivo são a função “*rgb_to_CIELAB*” (Figura 51) que converte uma cor do sistema RGB para o sistema CIE L*a*b, e a função “*get_all_app_screens_path*” (Figura 52) que retorna todos os caminhos das telas dos aplicativos em uma lista.

```
# Função que converte de uma cor do formato RGB para o formato CIE L*a*b
def rgb_to_CIELAB(rgb_color):
    color_sRGB = tuple((float(c)/255) for c in rgb_color)
    color_CIELAB = colour.XYZ_to_Lab(colour.sRGB_to_XYZ(color_sRGB))
    color = [color_CIELAB[0],color_CIELAB[1],color_CIELAB[2]]
    return color
```

Figura 51. Função “rgb_to_CIELAB” do arquivo “helper.py”

```
# Função que retorna uma lista contendo todos os caminhos das telas individualmente
def get_all_app_screens_path(universe_local, so, format):
    apps_un_path_dict = {}
    for path in Path(universe_local).rglob('*.'+format):
        path_split = ''
        if(so == 'windows'):
            path_split = str(path).split('\\')
        if(so == 'linux'):
            path_split = str(path).split('/')
        app = path_split[(len(path_split) - 1)]
        app = str(app).split('_')[0]
        apps_un_path_dict.setdefault(app, []).append(str(path))
    return apps_un_path_dict
```

Figura 52. Função “get_all_app_screens_path” do arquivo “helper.py”

5. CONCLUSÃO

Por meio desse trabalho se propõe uma solução para a avaliação da originalidade da paleta de cores em aplicativos Android dentro do contexto de Educação Básica, um tema ainda pouco abordado atualmente.

Na fundamentação teórica, foi feita uma análise acerca dos assuntos relacionados com o tema, focando principalmente no conceito de originalidade de uma paleta de cores, na descrição do design de interface de usuário de aplicativos Android e em conceitos de visão computacional para decidir o melhor método para avaliar a originalidade das paletas de cores.

Por meio de um mapeamento sistemático, foram levantados poucos trabalhos no estado da arte. Sendo que os três trabalhos obtidos ao final da revisão apresentaram pouco (ou em alguns casos nenhum) consenso para auxiliar nas decisões tomadas sobre os modelos de avaliação da originalidade das cores. No entanto, foi possível extrair algumas informações sobre os modelos de cores utilizados e sobre algumas formas de verificar a similaridade entre as cores.

Um modelo de avaliação da originalidade da paleta de cores em aplicativos foi desenvolvido, usando métodos de extração e análise que minimizam o impacto de trabalhar com uma quantidade de cores reduzida, sendo que a nota atribuída permanece coerente com a percepção humana. A maior dificuldade do trabalho estava em encontrar o ponto mais próximo possível da percepção humana para avaliar a originalidade da paleta de cores de um aplicativo. Na tentativa de alcançar a melhor aproximação diferentes fatores foram considerados (uso do sistema de cores CIE L*a*b, estudo da percepção humana para as paletas, considerar a porcentagem das cores na tela, parâmetros de ajuste e limite para cálculos, entre outros). Procurando minimizar os riscos de ameaças à validade, cada fator recebeu uma atenção especial para definir qual seria a abordagem mais adequada a ser adotada. O sistema CIE L*a*b foi escolhido por ser o sistema mais coerente com a percepção humana de cores que há atualmente. O estudo da percepção humana utilizou valores iniciais (de quantidade de cores e de porcentagem presente no aplicativo) pensados para facilitar a forma com que o ser humano julga as paletas. E os parâmetros para a avaliação entre um par de aplicativos foram especificados ao analisar um conjunto de exemplares utilizando diferentes valores.

Assim, os resultados obtidos na avaliação da solução proposta em comparação com as notas manualmente alocadas por humanos apontam resultados positivos. Aplicando a mesma metodologia para ambos os casos (notas manuais e notas atribuídas automaticamente pela solução), a diferença no resultado final é de apenas 0,5 ponto a

mais para as notas atribuídas pela solução, representando uma diferença aceitável, considerando o contexto de Educação Básica em que o tema está inserido.

Outra vantagem em utilizar o modelo proposto está no tempo total de execução. Mesmo ao executar a avaliação da originalidade da paleta do aplicativo “TesteFacil” para um universo de referência contendo 1773 aplicativos, o resultado foi gerado em aproximadamente 1 minuto e 30 segundos de execução, um valor inferior ao tempo normalmente levado por um avaliador humano para gerar a nota.

Dessa forma espera-se que o modelo desenvolvido neste TCC possa contribuir para apoiar a avaliação da originalidade de aplicativos desenvolvidos por estudantes da educação básica.

Trabalhos futuros incluem a implementação do módulo dentro da ferramenta Codemaster (Alves, 2019) para possibilitar o acesso online a esse tipo de apoio de avaliação. Também deverá ser feito um estudo de caso, aplicando o modelo proposto na prática, avaliando as paletas de aplicativos desenvolvidos por uma turma na educação básica.

REFERÊNCIAS

Alves, N. d. C., da. Codemaster: Um Modelo de Avaliação do Pensamento Computacional na Educação Básica Através da Análise de Código de Linguagem de Programação Visual. Dissertação (Mestrado) - Ciência da Computação, Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2019.

Alves, N. d. C., da., Gresse Von Wangenheim, C., Alberto, M., Martins-Pacheco, L. H. Uma Proposta de Avaliação da Originalidade do Produto no Ensino de Algoritmos e Programação na Educação Básica. In: Anais do Simpósio Brasileiro de Informática na Educação, Natal, Brasil, 2020.

Alves, N. d. C., Gresse Von Wangenheim, C., Martins-Pacheco, L.H. Assessing Product Creativity in Computing Education: A Systematic Mapping Study, Informatics in Education, 20(1), 19-45, 2021. https://doi.org/10.15388/infedu.2021.02

Agarwal, M., Maheshwari, R. P. Á trous gradient structure descriptor for content based image retrieval. In: International Journal of Multimedia Information Retrieval, vol 1, p. 129–138, 2012. https://doi.org/10.1007/s13735-012-0005-5.

Araújo, R.J, d., Reis, J.C, d., Bonacin, R. Colors Similarity Computation for User Interface Adaptation. In: Antona, M., Stephanidis, C. (eds) Universal Access in Human–Computer Interaction. Design and Development Approaches and Methods, vol 10277. Springer, Cham, 2017. https://doi.org/10.1007/978-3-319-58706-6_27

Brasil. Ministério da Educação. Conselho Nacional de Educação. Normas sobre Computação na Educação Básica – Complemento à BNCC. Processo Nº 23001.001050/2019-18, 2022.

Cavallo, D., Singer, H., Gomes, A., Bittencourt, I., Silveira, I. Inovação e Criatividade na Educação Básica: Dos conceitos ao ecossistema. Revista Brasileira de Informática na Educação, 24(2), 2016.

Chen, W., Shi, Y.Q., Xuan, G. Identifying Computer Graphics using HSV Color Model and Statistical Moments of Characteristic Functions. IEEE International Conference on Multimedia and Expo, 1123 - 1126, 2007. http://doi.org/10.1109/ICME.2007.4284852.

Datacolor. Color Differences & Tolerances: Commercial Color Acceptability. 2013. Disponível em: http://industrial.datacolor.com/support/wp-content/uploads/2013/01/Color-Differences-Tolerances.pdf. Acesso em: 20 de junho de 2022.

Goree, S., Doosti, B., Crandall, D., Makoto Su, N. Investigating the Homogenization of Web Design: A Mixed-Methods Approach. In Proc. of the CHI Conference on Human Factors in Computing Systems. ACM, New York, EUA, 1–14, 2021. https://doi.org/10.1145/3411764.3445156.

Gresse von Wangenheim, C., Hauck, J. C. R., Demetrio, M. F., Pelle, R. Alves, N. d. C., Barbosa, H., Azevedo, L. F. CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs. Informatics in Education, 17(1), 117-150, 2018.

Gresse von Wangenheim, C., Araújo Porto, J.V., Hauck, J.C.R., Borgatto, A.F. Do we agree on user interface aesthetics of Android apps? arXiv:1812.09049[cs.SE], 2018b.

Haddaway, N. R., Woodcock, P., Macura, B., Collins, A. Making literature reviews more reliable through application of lessons from systematic reviews. Conservation Biology, v. 29, issue 6, p. 1596-1605, 2015. https://doi.org/10.1111/cobi.12541.

Hamuda, E., Ginley, B. M., Glavin, M., Jones, E. Improved image processing-based crop detection using Kalman filtering and the Hungarian algorithm, Computers and Electronics in Agriculture, v 148, p 37-44, 2018. https://doi.org/10.1016/j.compag.2018.02.027.

Ibraheem, N., Hasan, M., Khan, R. Z. Understanding Color Models: A Review. ARPN Journal of Science and Technology, 2012.

Johnson-Eilola, J., Selber A. S. Plagiarism, originality, assemblage. Computers and Composition, 24(4), p 375-403, 2007. https://doi.org/10.1016/j.compcom.2007.08.003.

Jyoti Bora, D., Kumar Gupta, A. Effect of Different Distance Measures on the Performance of K-Means Algorithm: An Experimental Study in Matlab. International Journal of Computer Science and Information Technologies, 5, 2014.

Khokher, A., Talwar, R. A fast and effective image retrieval scheme using color-, texture-, and shape-based histograms. Multimedia Tools and Applications, India, 2016. https://doi.org/10.1007/s11042-016-4096-5.

Kuhn, H. W. A tale of three eras: The discovery and rediscovery of the Hungarian Method. European Journal of Operational Research, vol 219, issue 3, p 641-651, 2012. https://doi.org/10.1016/j.ejor.2011.11.008.

Lima, A. L. de. S. Gresse von Wangenheim, C. Assessing the Visual Esthetics of User Interfaces: A Ten-Year Systematic Mapping. International Journal of Human–Computer Interaction, 2021. https://doi.org/10.1080/10447318.2021.1926118.

Lima, A. L. de. S., Alves, N. d. C. Roteiro para coleta de código e screenshots de apps App Inventor. Comunicação interna, CnE/INCOD/INE/UFSC, 2020.

Loesdau, M., Chabrier, S., Gabillon, A. Hue and Saturation in the RGB Color Space. In: Elmoataz, A., Lezoray, O., Nouboud, F., Mammass, D. (eds) Image and Signal Processing. Lecture Notes in Computer Science, vol 8509. Springer, Cham, 2014. https://doi.org/10.1007/978-3-319-07998-1_23.

Lytle, N., Cateté, V., Boulden, D., Dong, Y., Houchins, J., Milliken, A., Isvik, A., Bounajim, D., Wiebe, E., Barnes, T. Use, Modify, Create: Comparing Computational Thinking Lesson Progressions for STEM Classes. In Proc. of the ACM Conference on Innovation and Technology in Computer Science Education . ACM, New York, EUA, 395–401, 2019. https://doi.org/10.1145/3304221.3319786.

Melo da Silva, D. Suporte a Unidade Instrucional de Desenvolvimento de Aplicativos com Técnicas de UX Design para o Ensino Básico. Universidade Federal de Santa Catarina, Centro Tecnológico, Departamento de Informática e Estatística, 2017.

Mishra, P., Henriksen, D. A NEW Approach to Defining and Measuring Creativity: Rethinking Technology & Creativity in the 21st Century. TECH TRENDS, 57, 10-13, 2013. https://doi.org/10.1007/s11528-013-0685-6

Patton E.W., Tissenbaum M., Harunani F. MIT App Inventor: Objectives, Design, and Development. In: Kong SC., Abelson H. (eds) Computational Thinking Education. Springer, Singapore, 2019. https://doi.org/10.1007/978-981-13-6528-7_3

Pardede, J. Sitohang, B. Akbar, S. Khodra, M. L. Comparison of Similarity Measures in HSV Quantization for CBIR. International Conference on Data and Software Engineering (ICoDSE), p. 1-6, 2017. https://doi.org/10.1109/ICODSE.2017.8285854.

Piasecki, J., Waligora, M., Dranseika, V. Google Search as an Additional Source in Systematic Reviews. Sci Eng Ethics 24, p. 809-810, 2018. https://doi.org/10.1007/s11948-017-0010-4

Pypi. Extcolors 1.0.0. 2020. Disponível em: https://pypi.org/project/extcolors/. Acesso em: 10 de outubro de 2022.

Python. Timeit - Measure execution time of small code snippets. 2022 Disponível em: https://docs.python.org/3/library/timeit.html. Acesso em: 12 de novembro de 2022.

Schlatter, T., Levinson, D. Visual usability: Principles and practices for designing digital applications. Morgan Kaufmann, 2013.

Schuessler, Z. Delta E 101. Disponível em: https://zschuessler.github.io/DeltaE/learn. Acesso em: 11 de junho de 2022.

Solecki, I,, d. S. Uma abordagem para a avaliação do design visual de aplicativos móveis criados com linguagens de programação baseadas em blocos. Universidade Federal de Santa Catarina, Centro Tecnológico, Departamento de Informática e Estatística, 2020.

Tang, Y., Hou, L. U., Cai, L., Mamoulis, N., Cheng, R. Earth Mover's Distance based Similarity Search at Scale. Proceedings of the VLDB Endowment, Hangzhou, China, 2014.

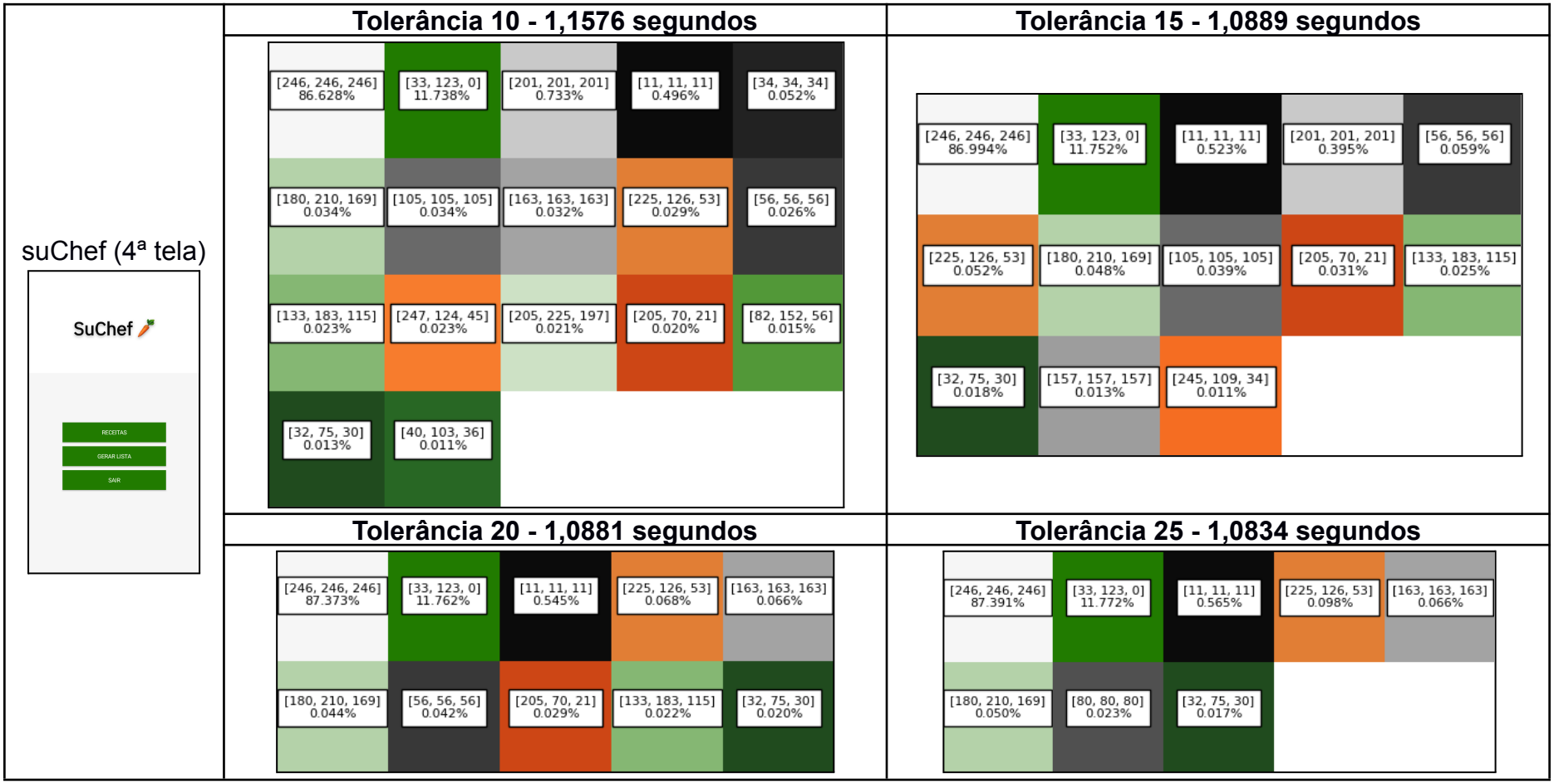
Yadav, A.,Cooper, S. Fostering Creativity Through Computing. Communications of the ACM, 60(2), 31-33, 2017.

Wolber, D., Abelson, H., Friedman, M. Democratizing Computing with App Inventor. GetMobile, 18(4), 2014.

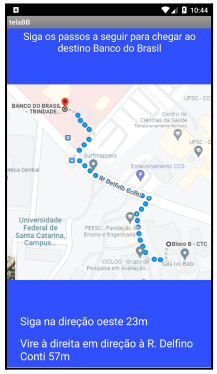
Zenggang, X., Zhiwen, T., Xiaowen, C., Xue-min, Z., Kaibin, Z., Conghuan, Y. Research on Image Retrieval Algorithm Based on Combination of Color and Shape Features. Journal of Signal Processing Systems, 93, 139–146, 2021. https://doi.org/10.1007/s11265-019-01508-y

ANEXO A

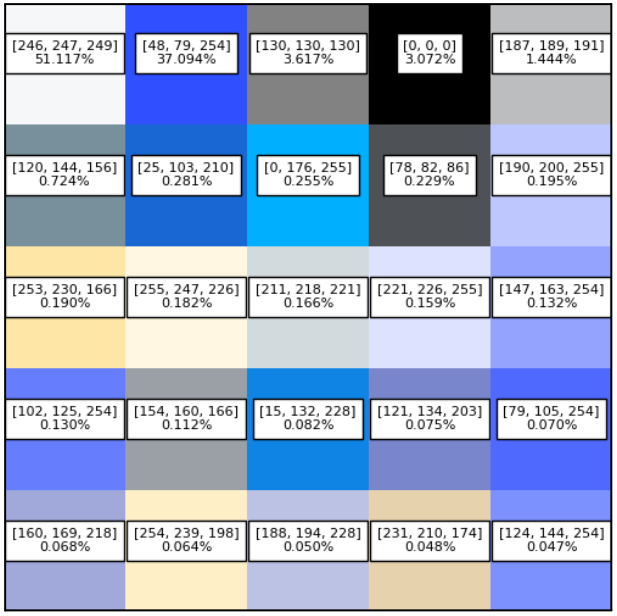
Paletas de cores extraídas para diferentes valores de tolerância (limitadas à 25 cores para demonstração), contendo o tempo total para extração e as porcentagens que cada cor ocupa na tela.



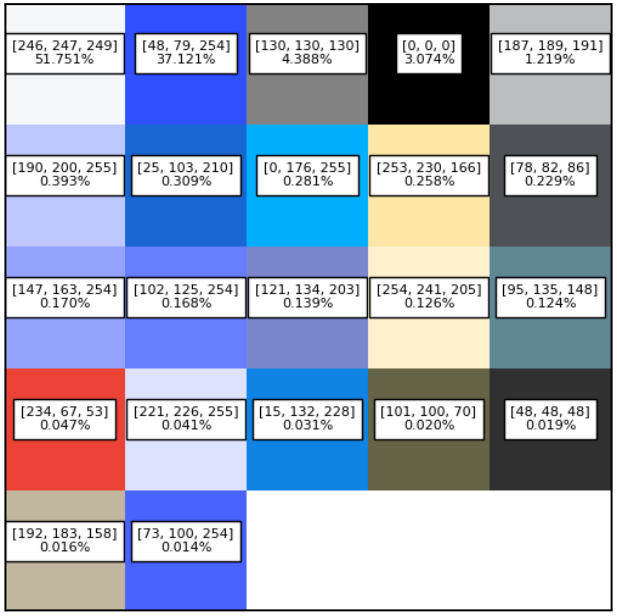
RotasSeguras
(2ª tela)



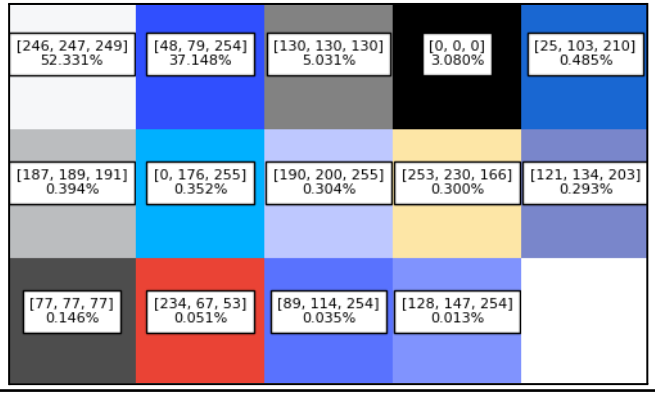
Tolerância 10 - 1,5653 segundos



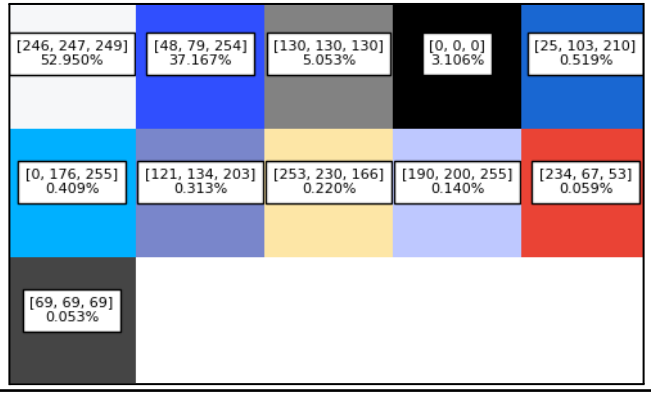
Tolerância 15 - 1,3033 segundos



Tolerância 20 - 1,2616 segundos

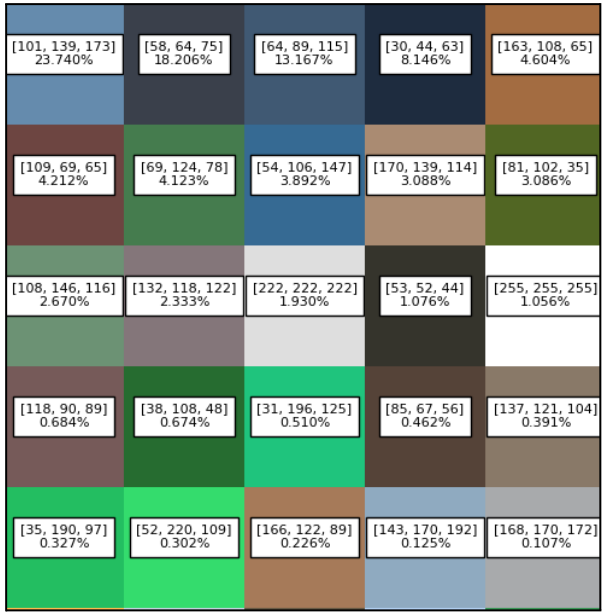


Tolerância 25 - 1,2296 segundos

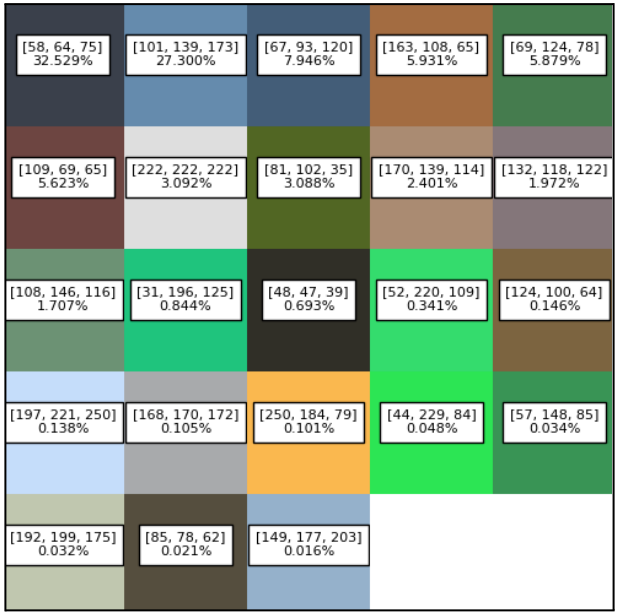


Meteorito (1ª tela)

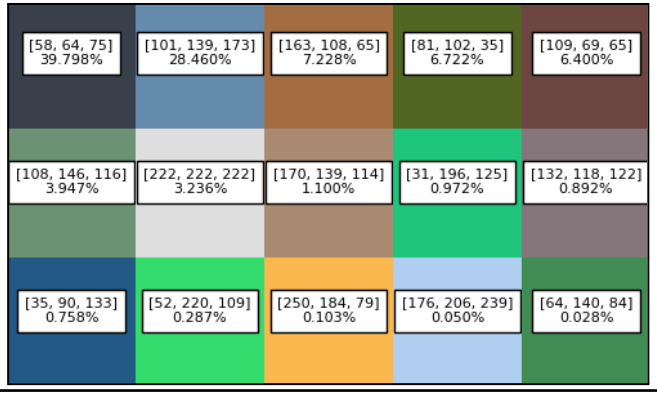
Tolerância 10 - 2,5544 segundos



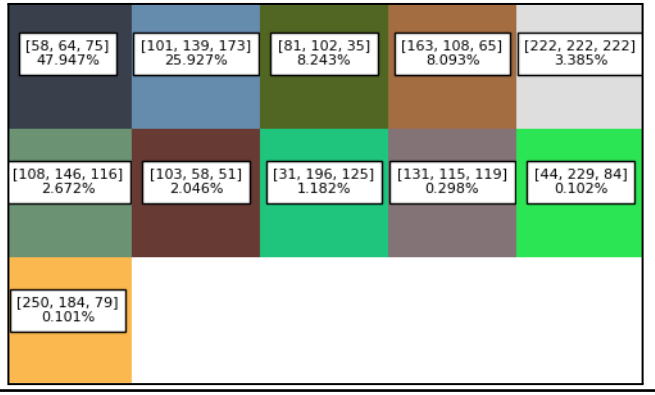
Tolerância 15 - 1,8885 segundos



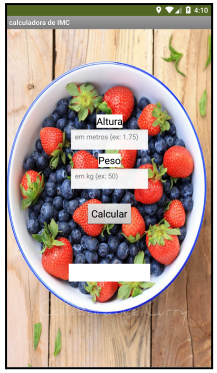
Tolerância 20 - 1,7593 segundos



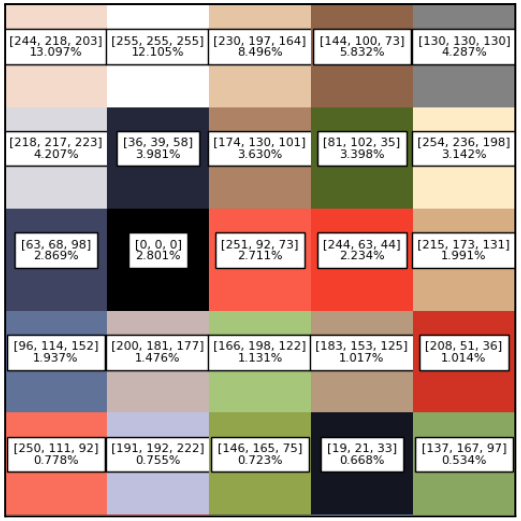
Tolerância 25 - 1,6244 segundos



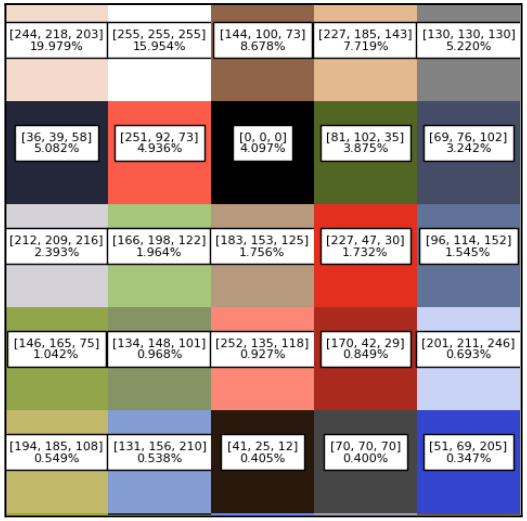
IMC (1ª tela)



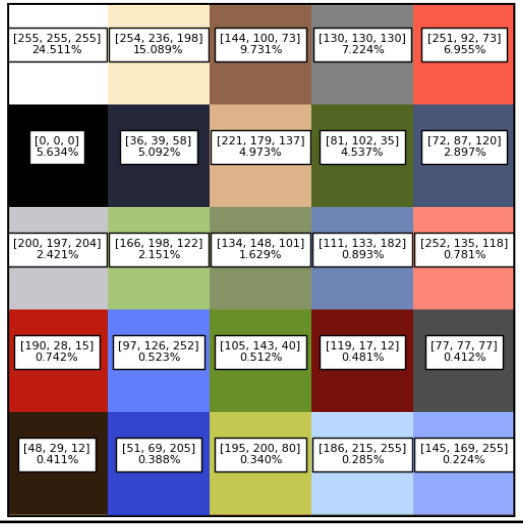
Tolerância 10 - 29,9575 segundos



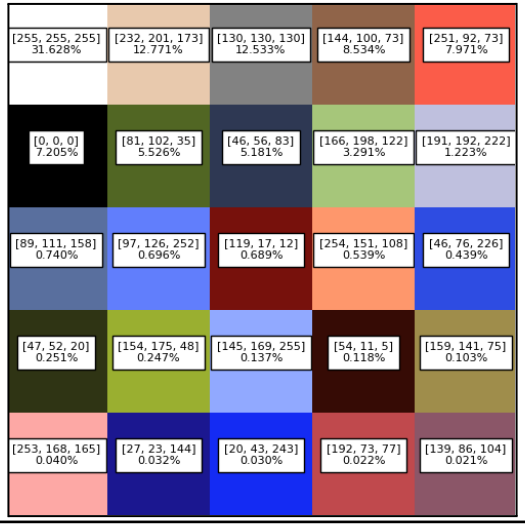
Tolerância 15 - 13,7138 segundos



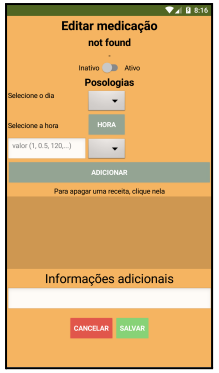
Tolerância 20 - 9,7474 segundos



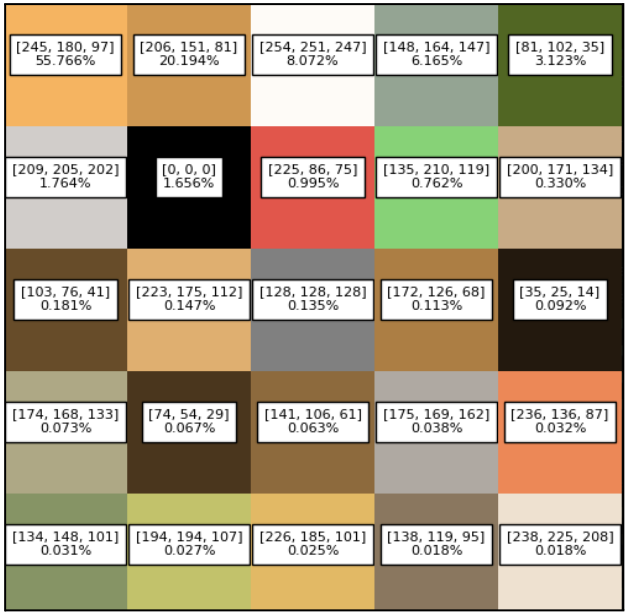
Tolerância 25 - 6,6271 segundos



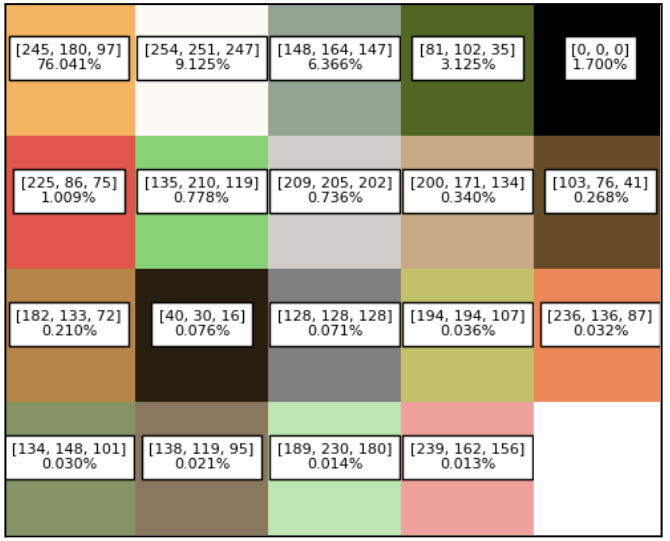
Medike (6ª tela)



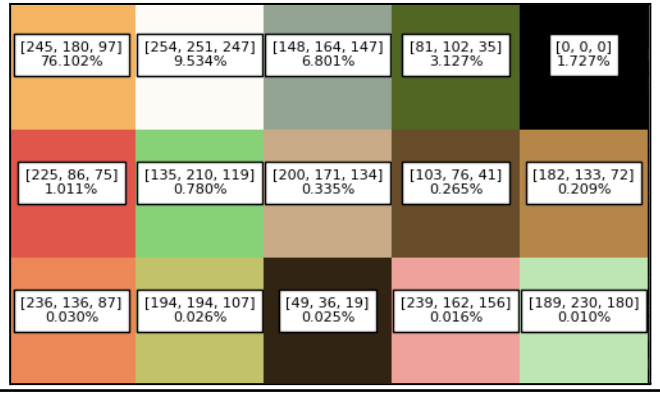
Tolerância 10 - 1,0159 segundos



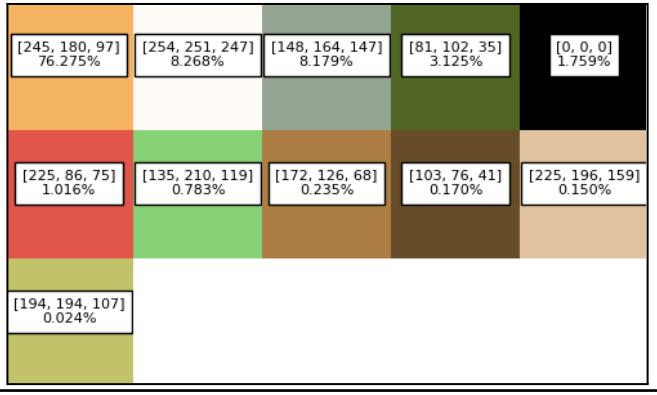
Tolerância 15 - 1,0463 segundos



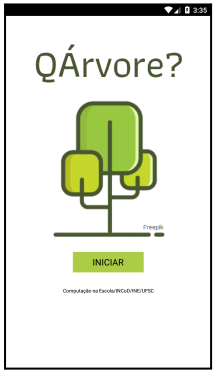
Tolerância 20 - 1,0355 segundos



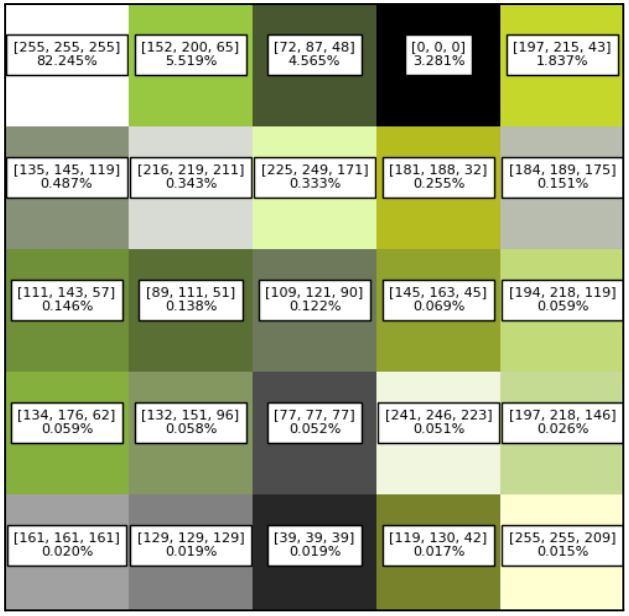
Tolerância 25 - 1,0425 segundos



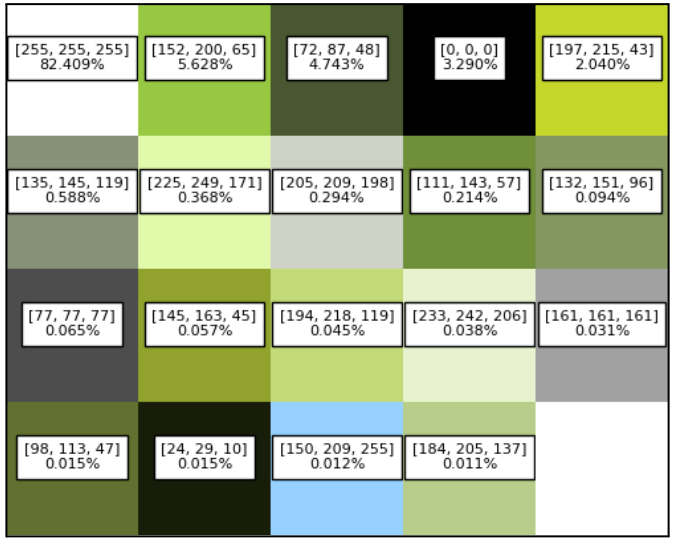
QArvore (1ª tela)



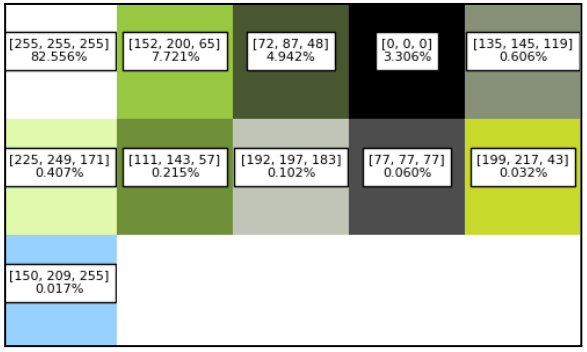
Tolerância 10 - 1,12 segundos



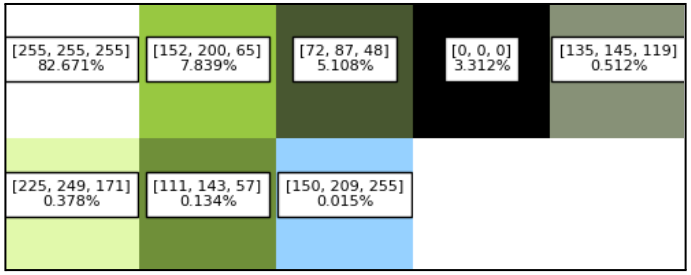
Tolerância 15 - 1,1129 segundos



Tolerância 20 - 1,081 segundos

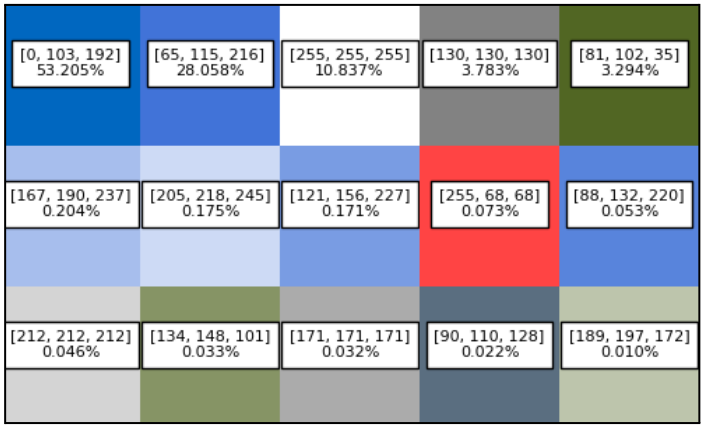


Tolerância 25 - 1,0626 segundos

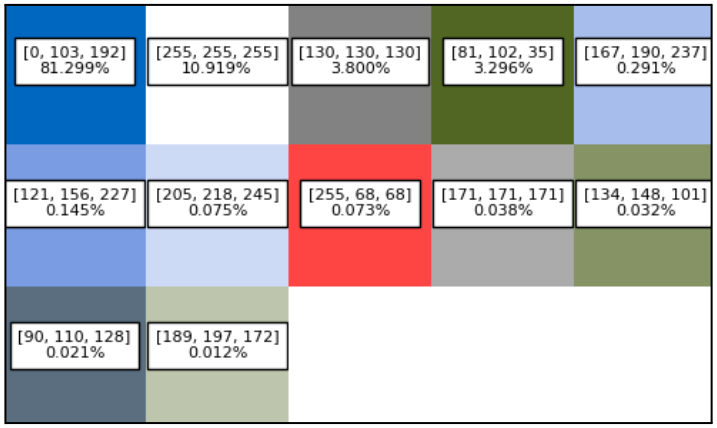


WeLearn (2ª tela)

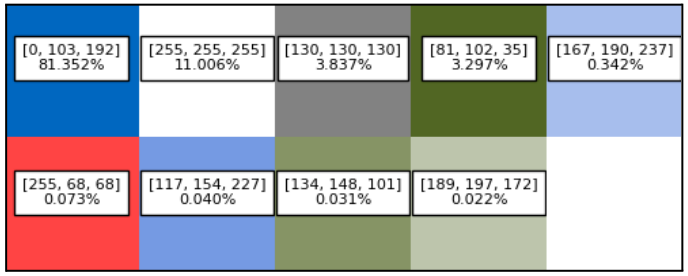
Tolerância 10 - 1,0133 segundos



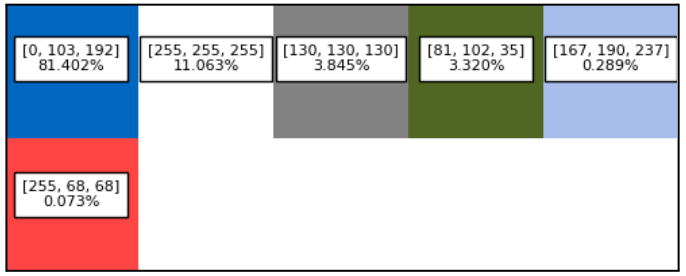
Tolerância 15 - 1,0362 segundos



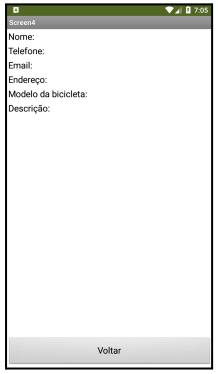
Tolerância 20 - 1,0394 segundos



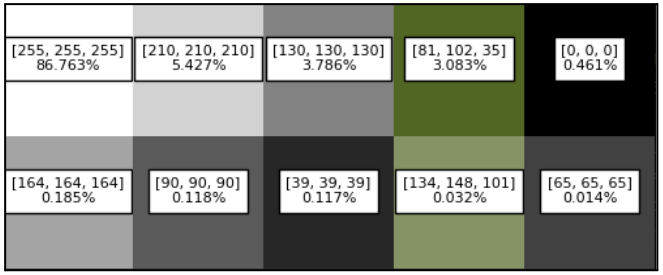
Tolerância 25 - 0,5024 segundos



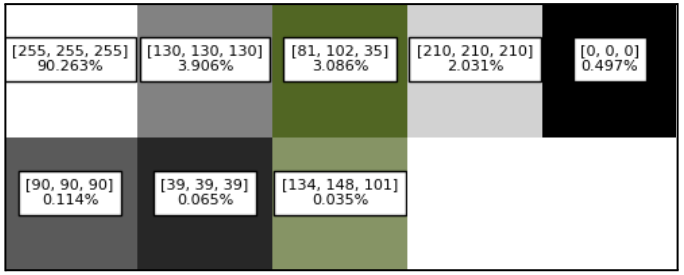
vaiDeBike (4ª tela)



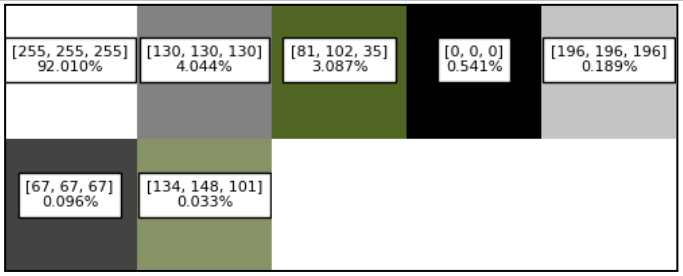
Tolerância 10 - 0,9958 segundos



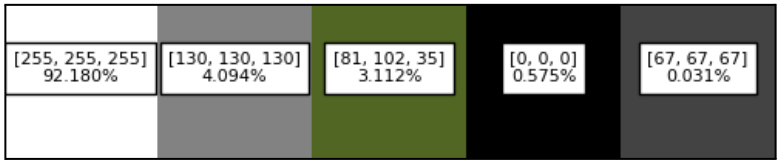
Tolerância 15 - 1,0152 segundos



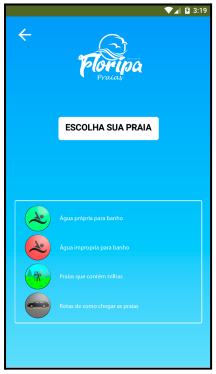
Tolerância 20 - 1,0234 segundos



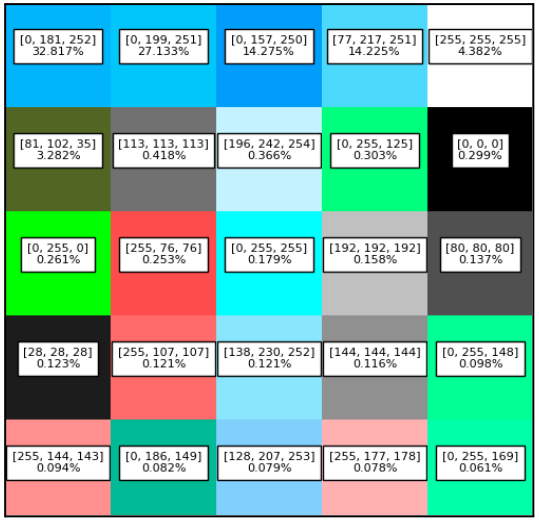
Tolerância 25 - 1,0085 segundos



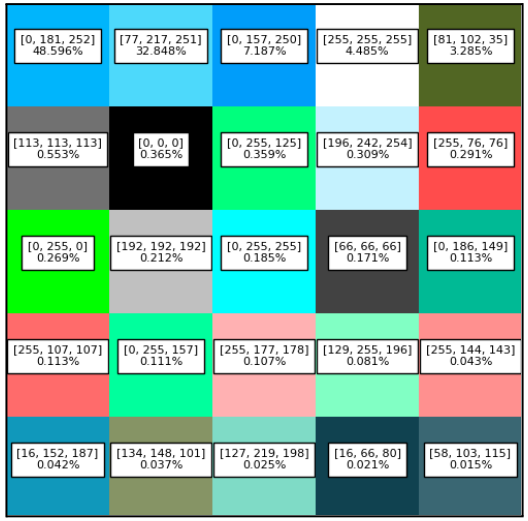
FloripaPraias
(7ª tela)



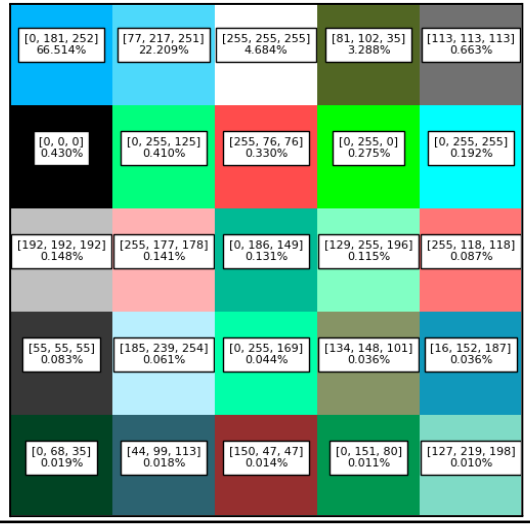
Tolerância 10 - 1,5147 segundos



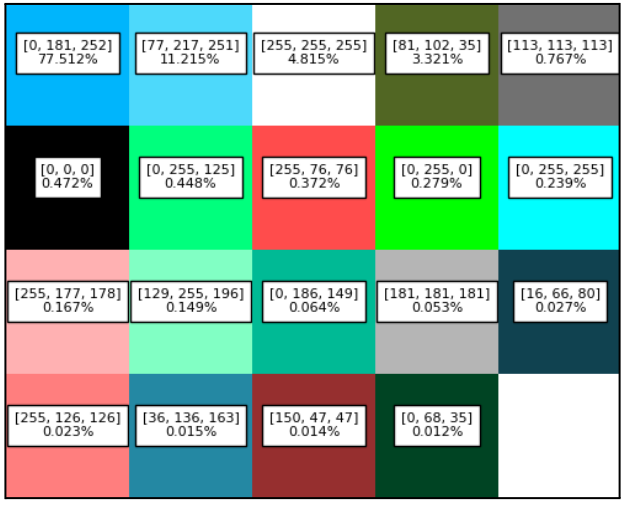
Tolerância 15 - 1,3324 segundos



Tolerância 20 - 1,2659 segundos

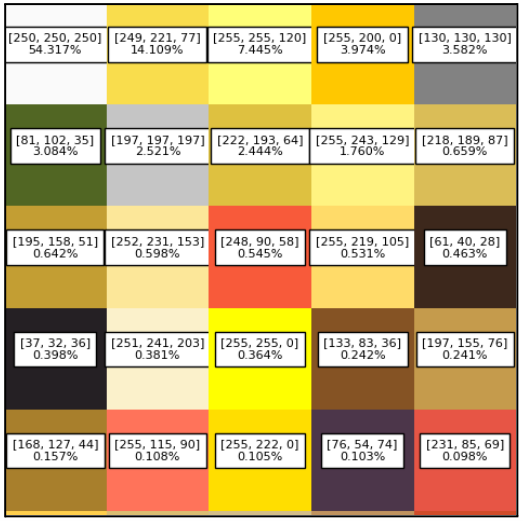


Tolerância 25 - 1,1805 segundos

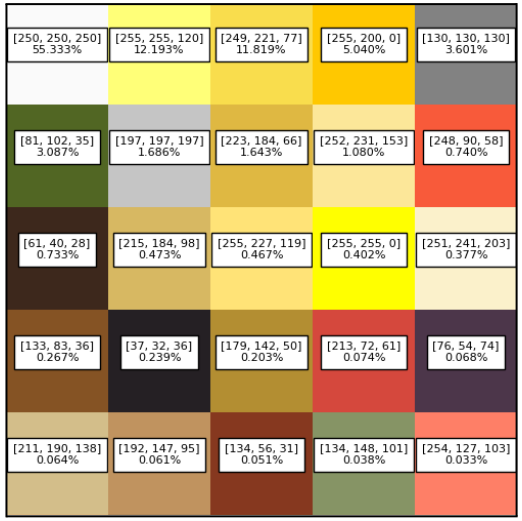


Pikachu (1ª tela)

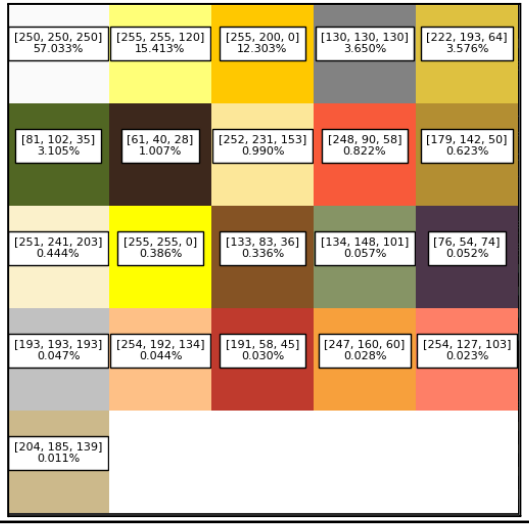
Tolerância 10 - 3,3529 segundos



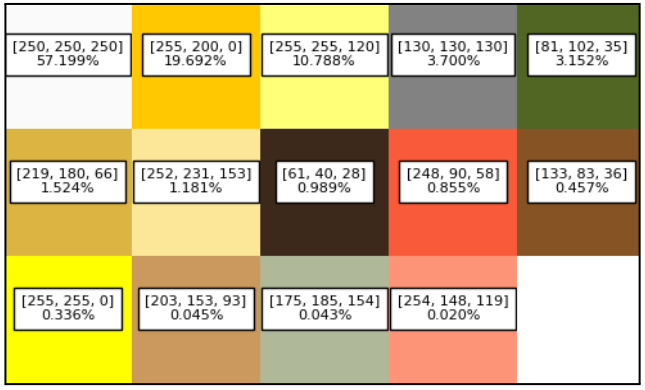
Tolerância 15 - 2,5846 segundos



Tolerância 20 - 2,2949 segundos

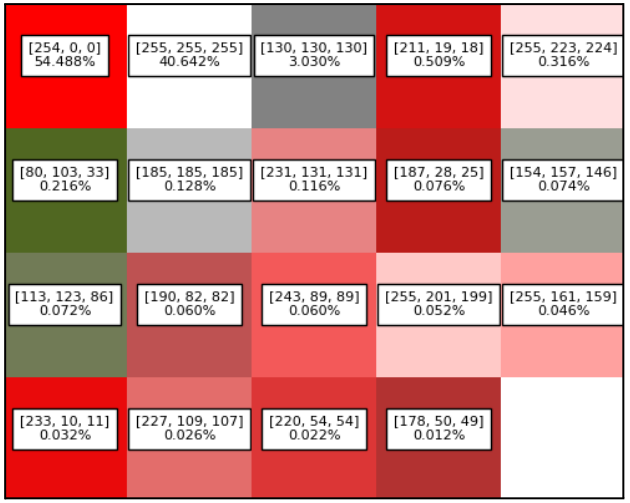


Tolerância 25 - 1,8425 segundos

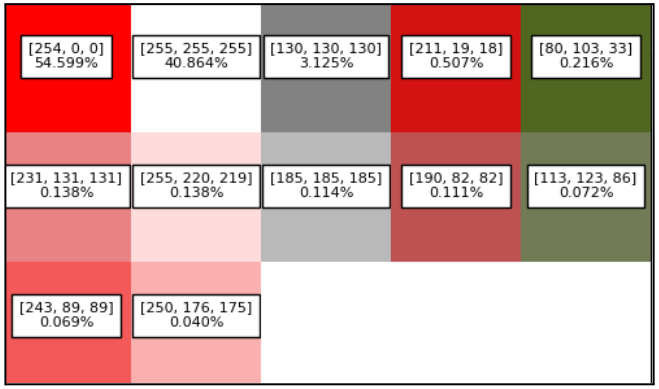


sos (1ª tela)

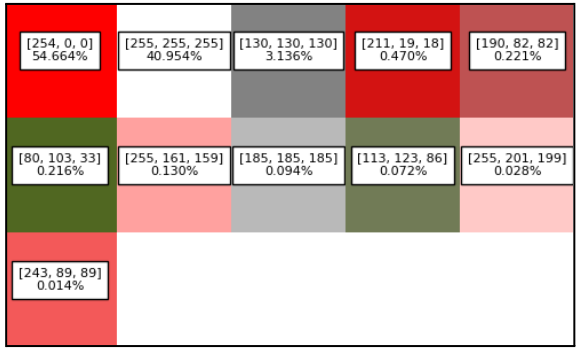
Tolerância 10 - 0,5446 segundos



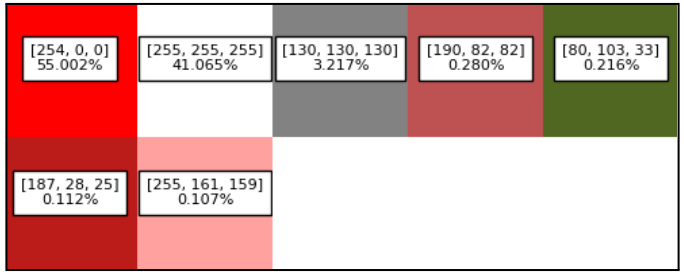
Tolerância 15 - 0,5374 segundos



Tolerância 20 - 0,5471 segundos

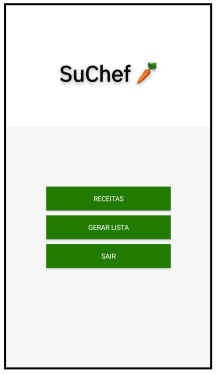
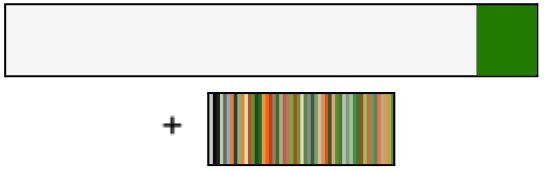
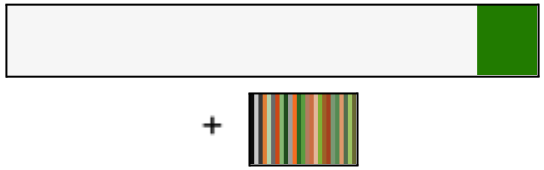
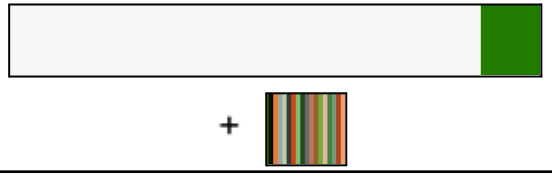
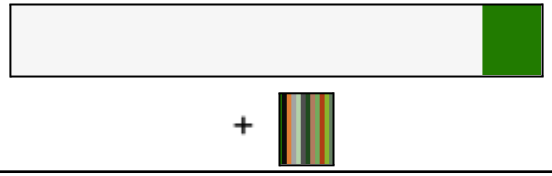
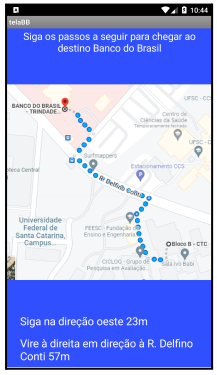
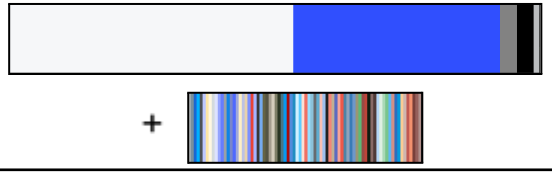
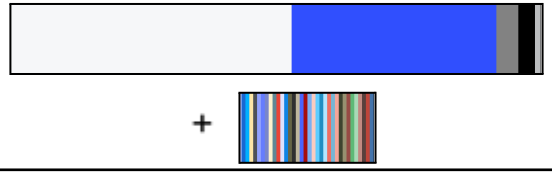
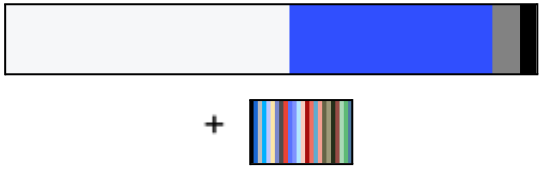
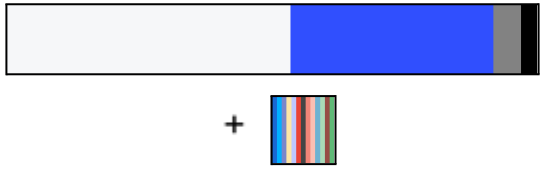


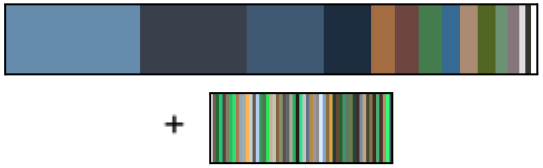
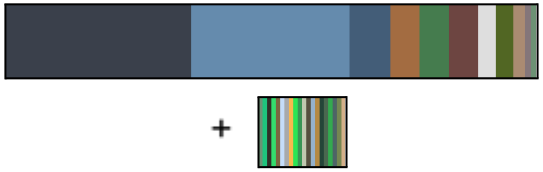
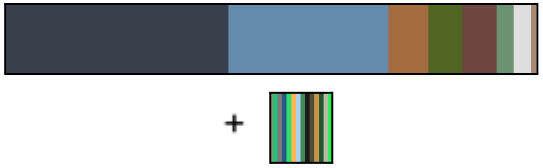
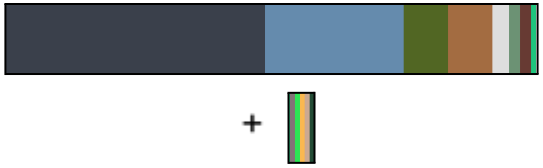
Tolerância 25 - 0,5024 segundos

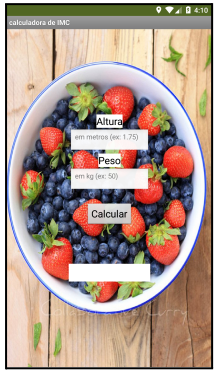
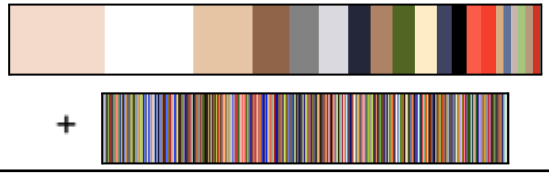
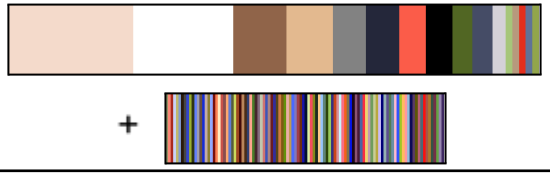
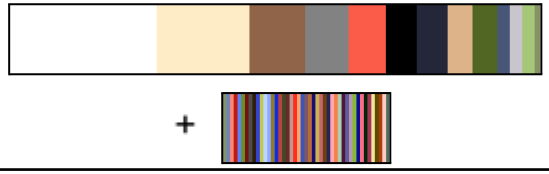
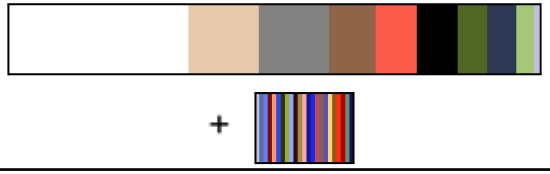


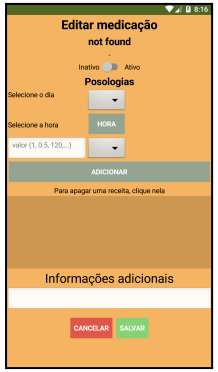
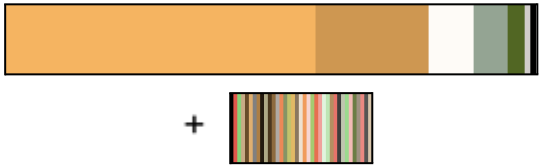
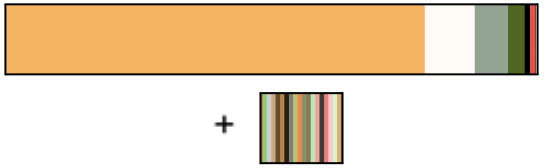
ANEXO B

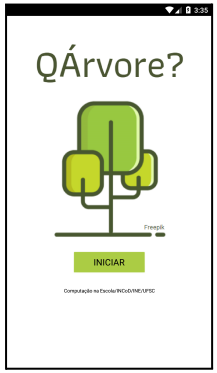
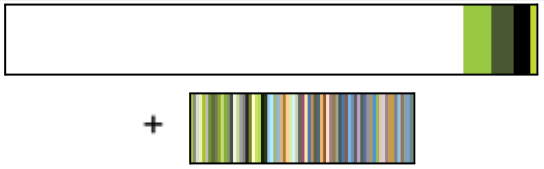
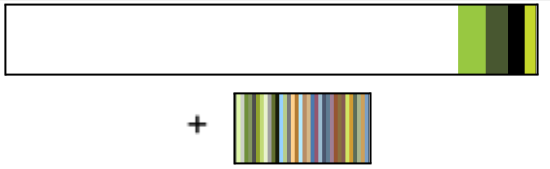
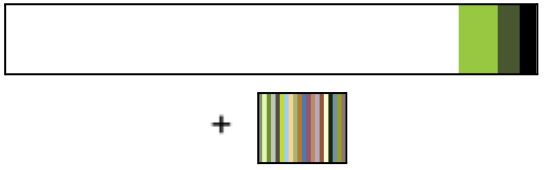
Paletas de cores proporcionais com porcentagem superior à 1% (para fins de demonstração) extraídas para diferentes valores de tolerância, contendo o total de cores extraídas e as demais cores logo abaixo.

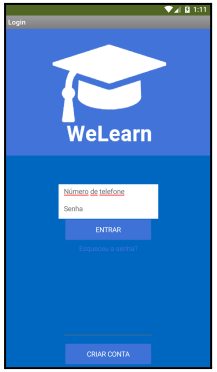
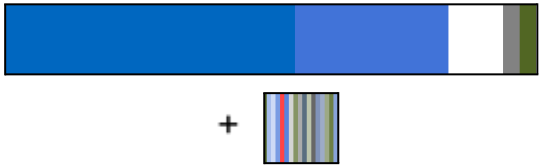
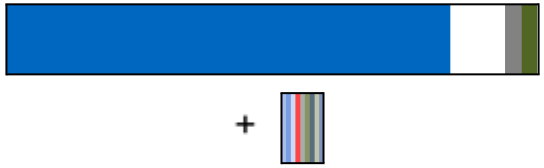
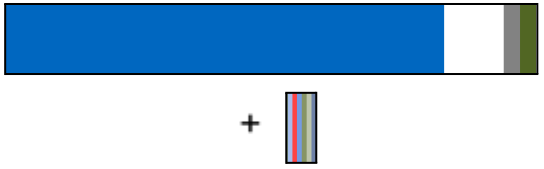
<table>
<tr><td rowspan="2">suChef (4ª tela)
 
</td><td>Tolerância 10 (63 cores)
  </td><td>Tolerância 15 (30 cores)
  </td></tr>
<tr><td>Tolerância 20 (21 cores)
  </td><td>Tolerância 25 (14 cores)
  </td></tr>
<tr><td rowspan="2">RotasSeguras (2ª tela)
 
</td><td>Tolerância 10 (85 cores)
  </td><td>Tolerância 15 (42 cores)
  </td></tr>
<tr><td>Tolerância 20 (28 cores)
  </td><td>Tolerância 25 (17 cores)
  </td></tr>
</table>

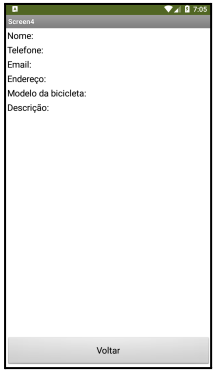
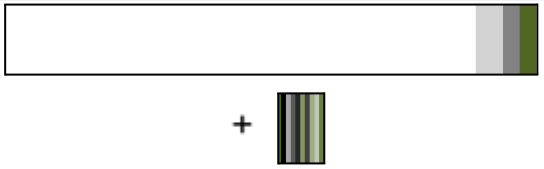
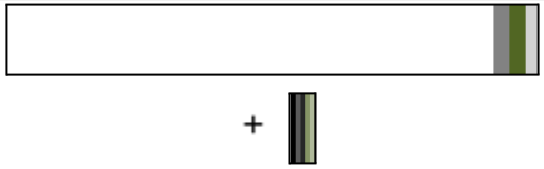
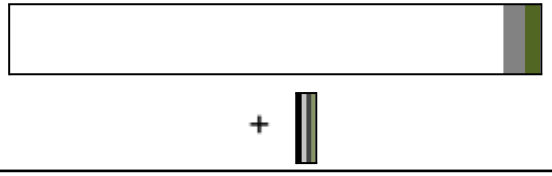
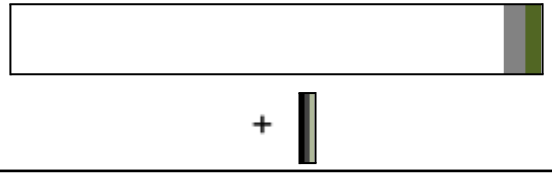
Meteorito (1ª tela)	Tolerância 10 (70 cores)	Tolerância 15 (30 cores)
		
	Tolerância 20 (21 cores)	**Tolerância 25 (13 cores)**
		

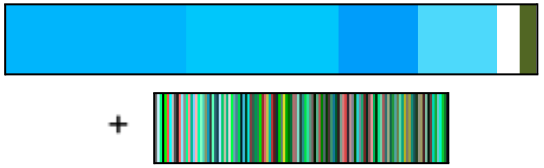
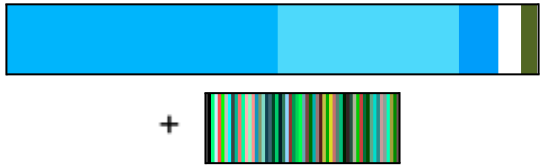
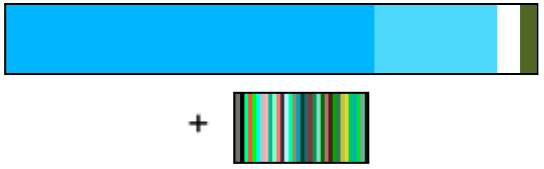
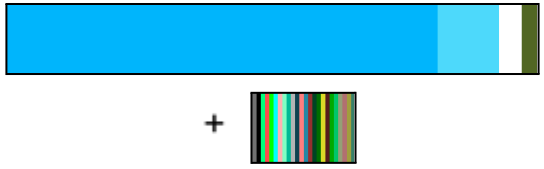
IMC (1ª tela)	Tolerância 10 (327 cores)	Tolerância 15 (125 cores)
		
	Tolerância 20 (57 cores)	**Tolerância 25 (32 cores)**
		

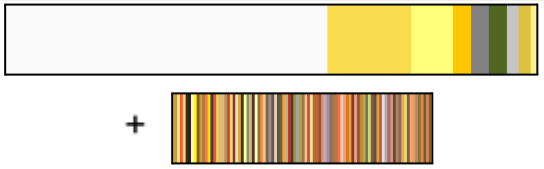
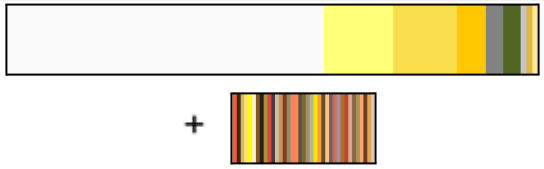
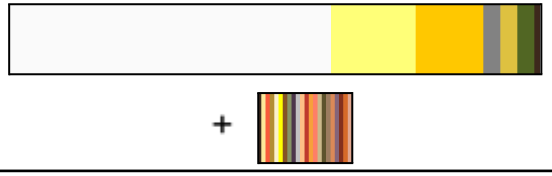
Medike (6ª tela)	Tolerância 10 (43 cores)	Tolerância 15 (24 cores)
		
	Tolerância 20 (17 cores)	**Tolerância 25 (14 cores)**
		

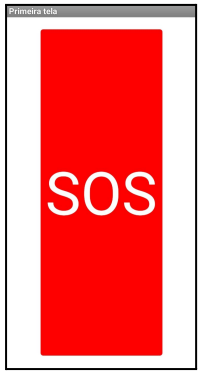
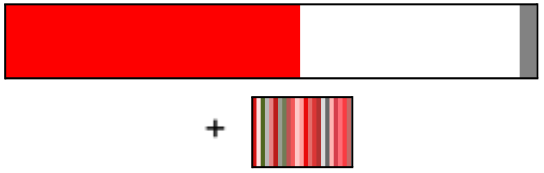
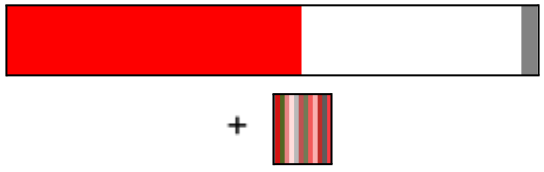
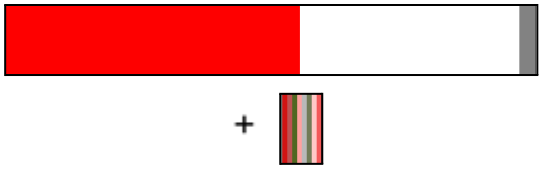
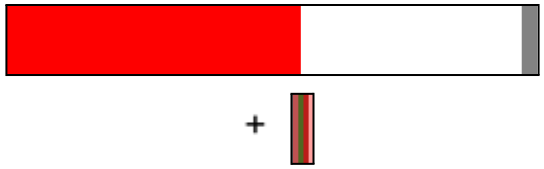
QArvore (1ª tela)	Tolerância 10 (80 cores)	Tolerância 15 (41 cores)
		
	Tolerância 20 (24 cores)	**Tolerância 25 (15 cores)**
		

WeLearn (2ª tela)	Tolerância 10 (23 cores)	Tolerância 15 (13 cores)
		
	Tolerância 20 (10 cores)	**Tolerância 25 (8 cores)**
		

vaiDeBike (4ª tela)	Tolerância 10 (13 cores)	Tolerância 15 (9 cores)
		
	Tolerância 20 (7 cores)	**Tolerância 25 (6 cores)**
		

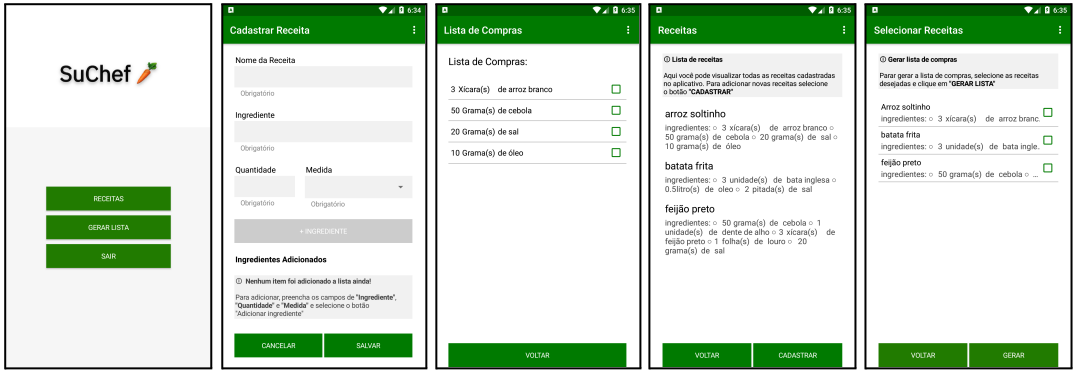
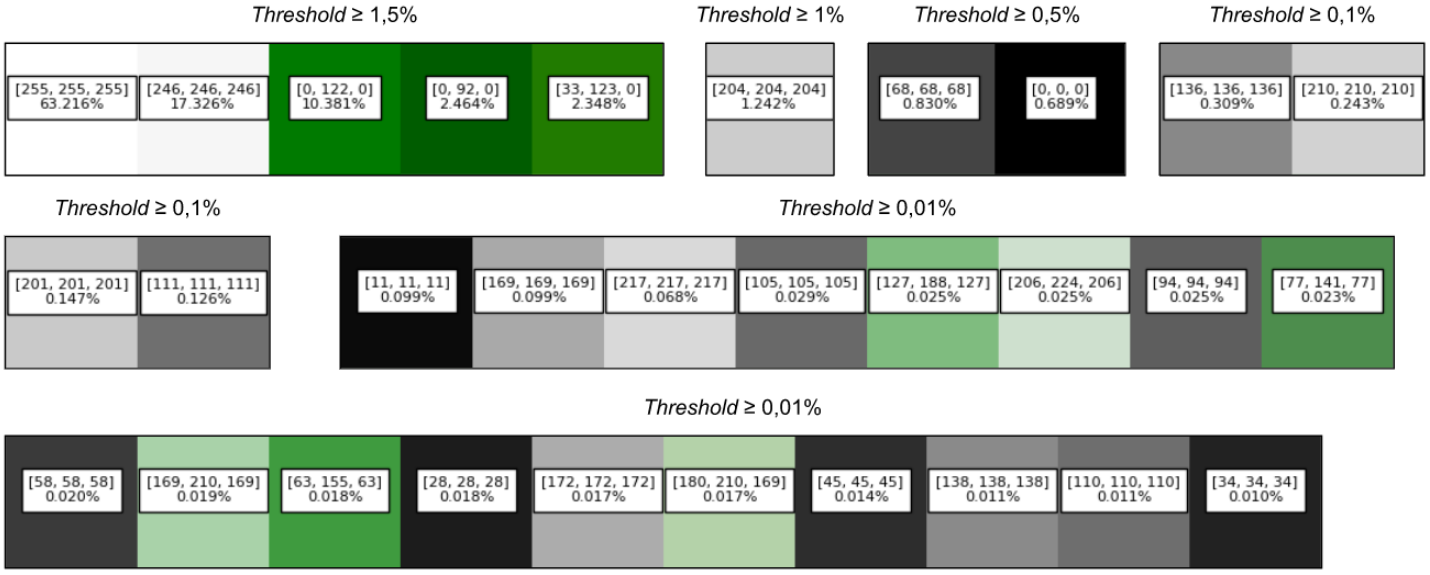
FloripaPraias (7ª tela) 	**Tolerância 10 (133 cores)** 	**Tolerância 15 (64 cores)** 
	Tolerância 20 (41 cores) 	**Tolerância 25 (29 cores)** 

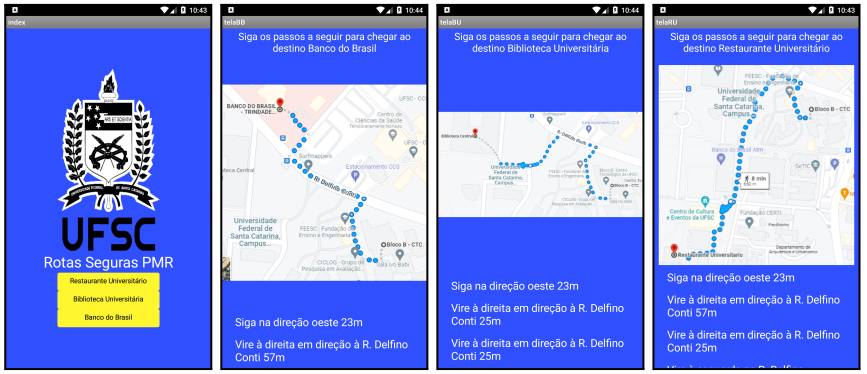
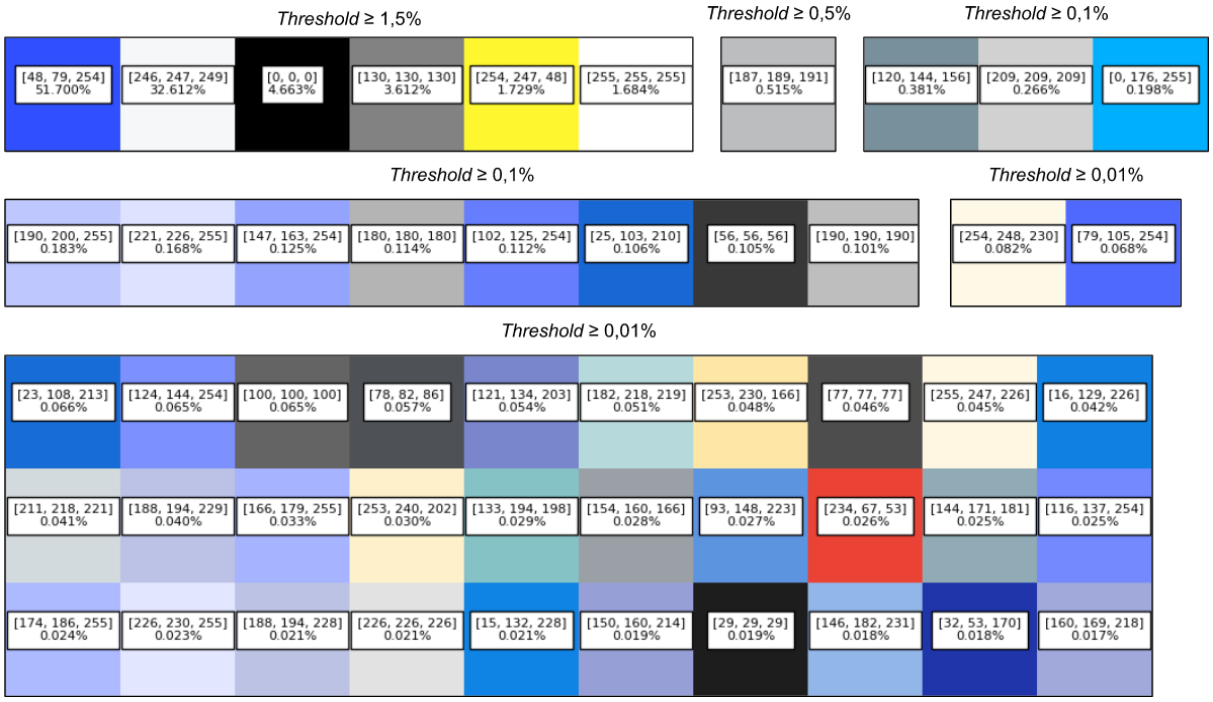
Pikachu (1ª tela) 	**Tolerância 10 (106 cores)** 	**Tolerância 15 (46 cores)** 
	Tolerância 20 (28 cores) 	**Tolerância 25 (17 cores)** 

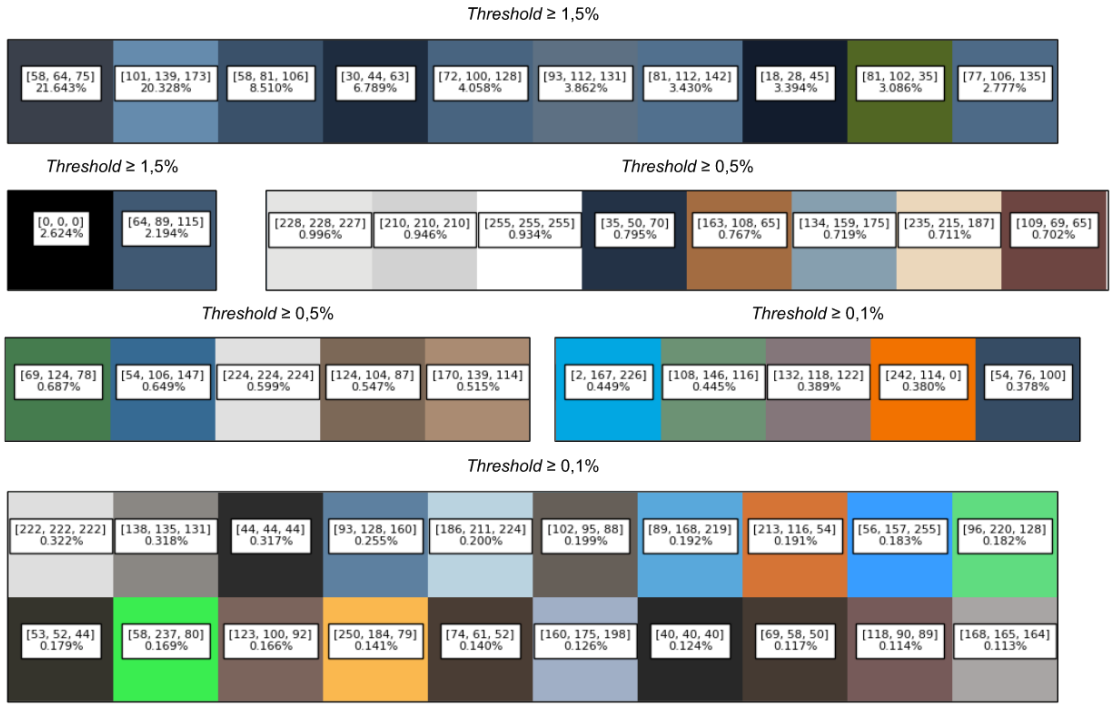
sos (1ª tela)	Tolerância 10 (26 cores)	Tolerância 15 (15 cores)
		
	Tolerância 20 (11 cores)	**Tolerância 25 (7 cores)**
		

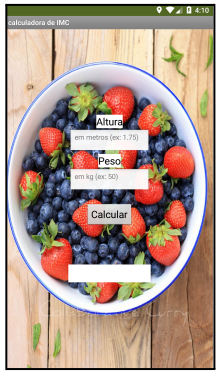
ANEXO C

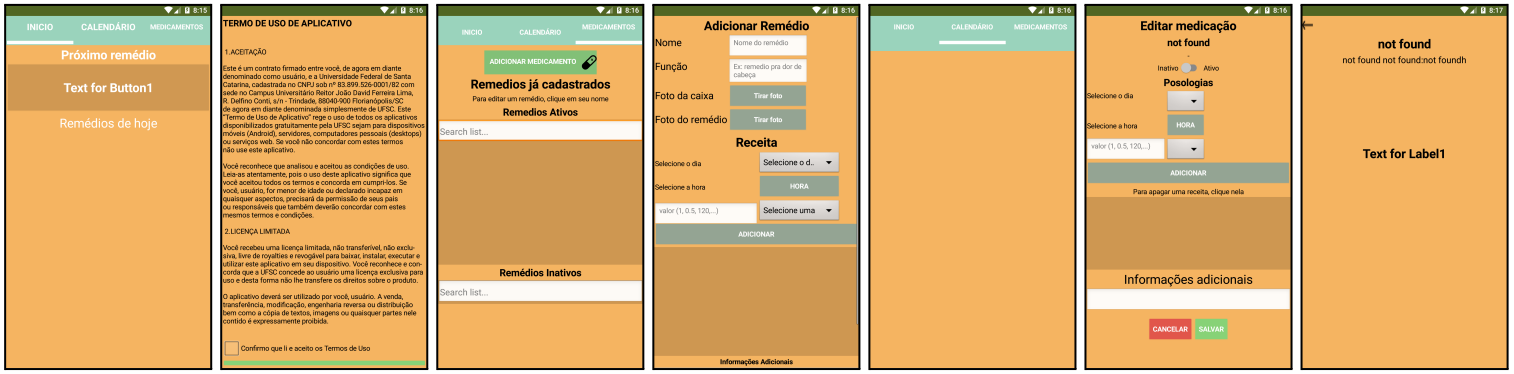
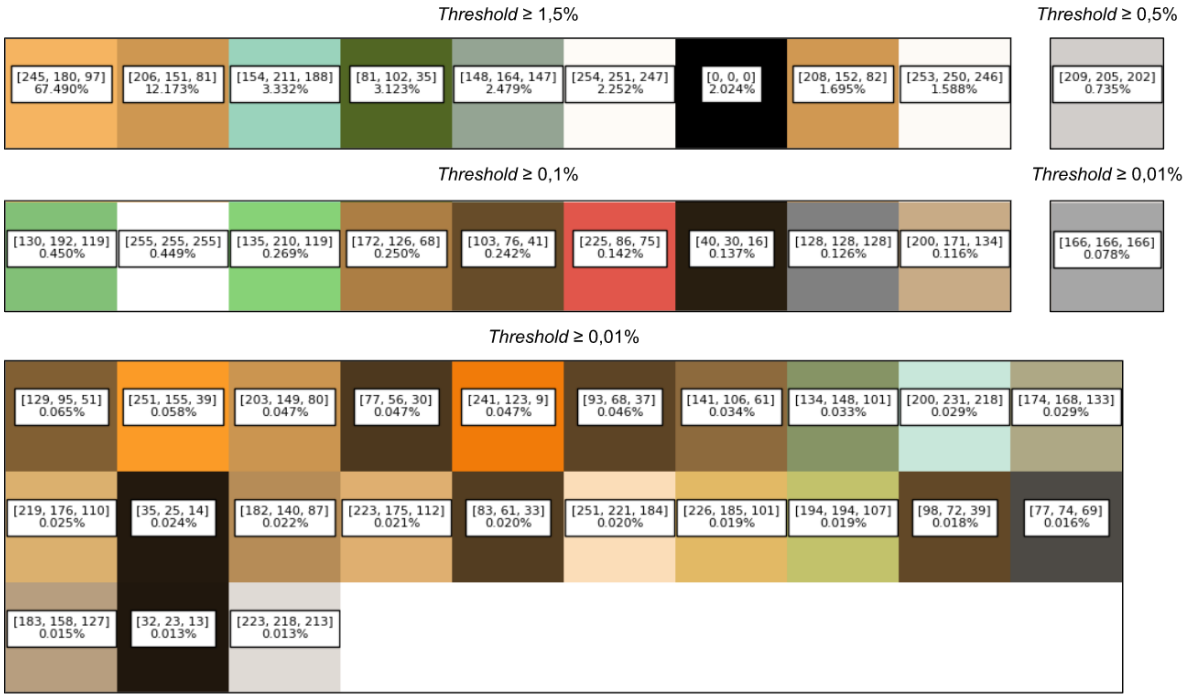
Comparação entre os valores de threshold dentro das paletas de diferentes aplicativos com valor de tolerância igual a 10

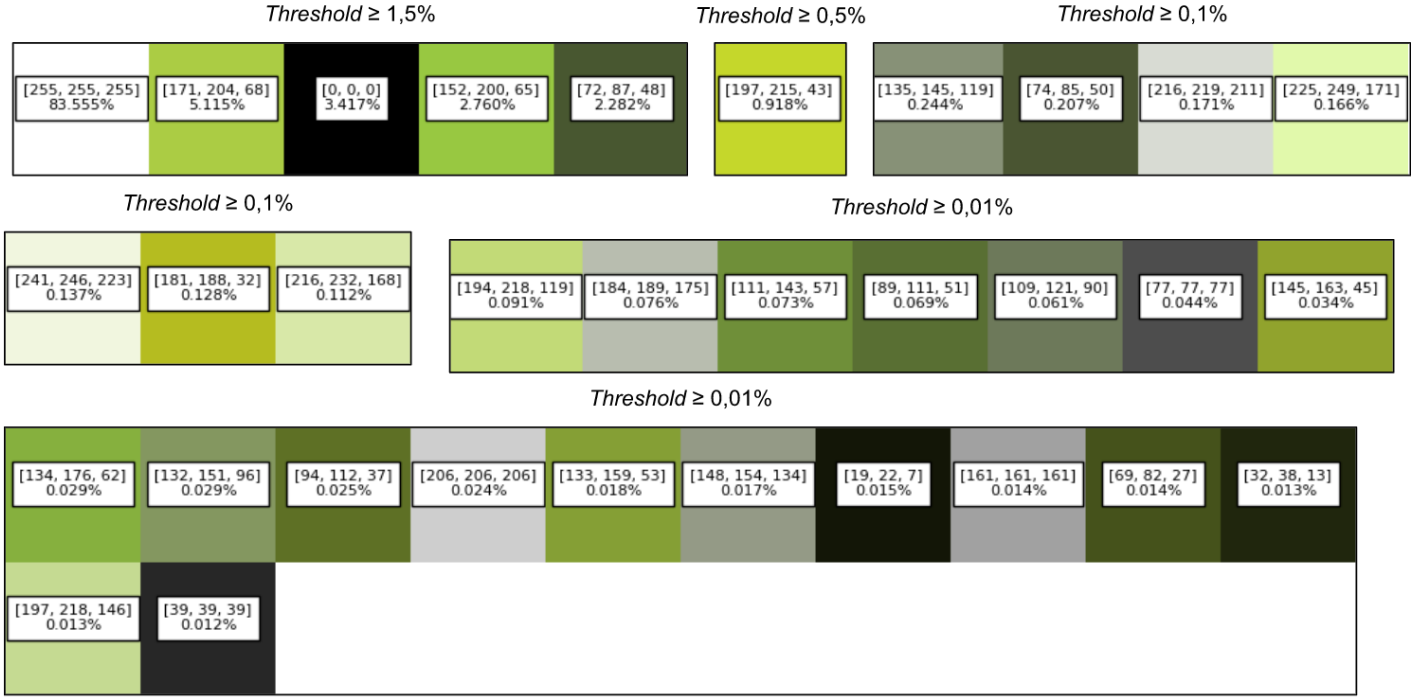
App “suChef”	
Threshold	

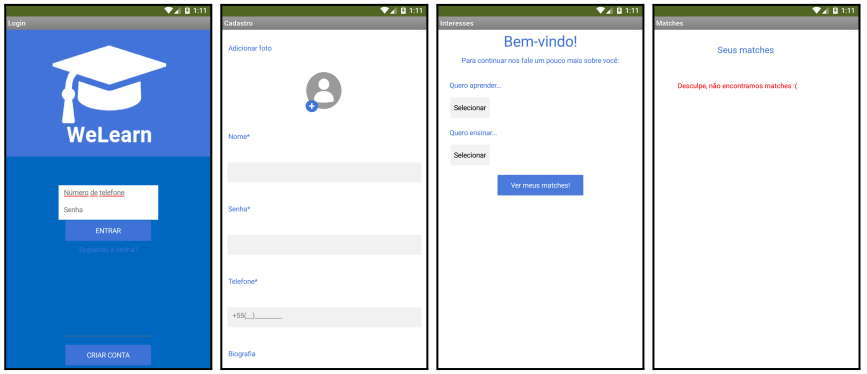
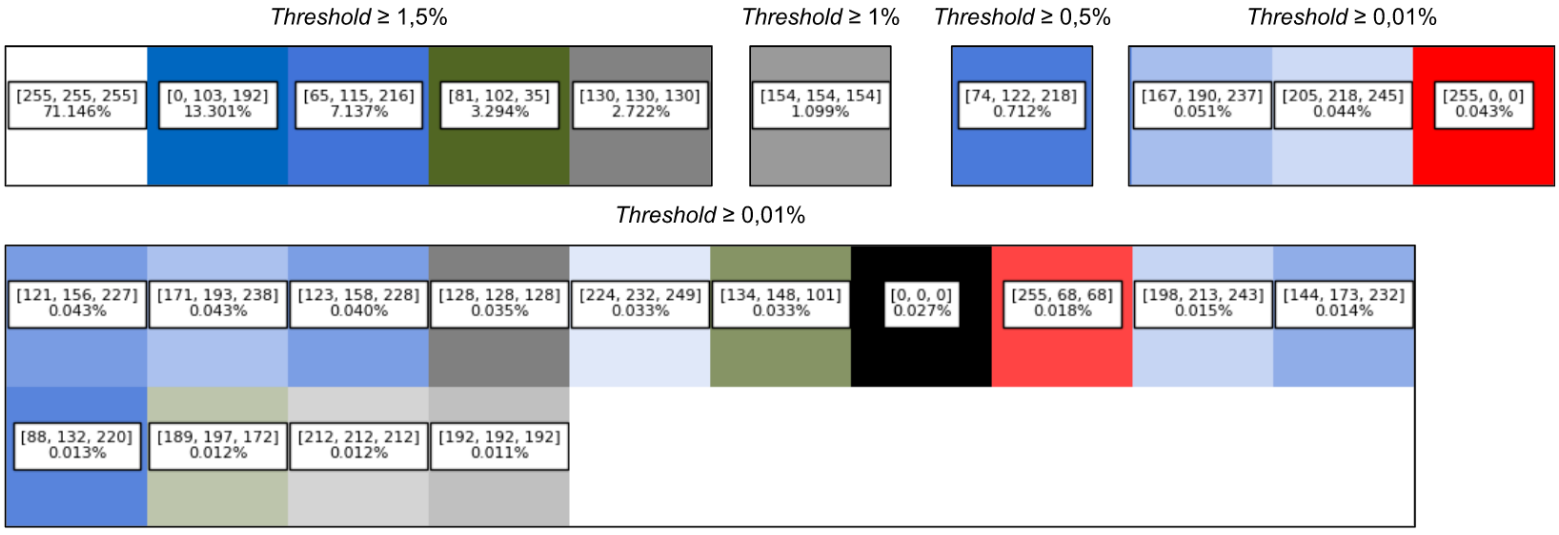
App “RotasSeguras ”	
Threshold	

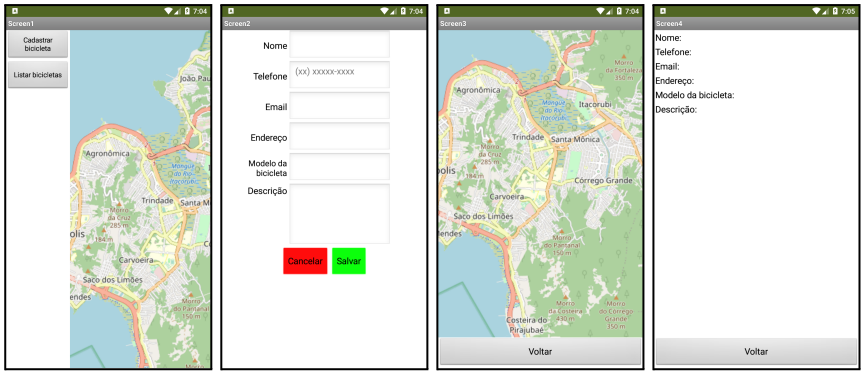
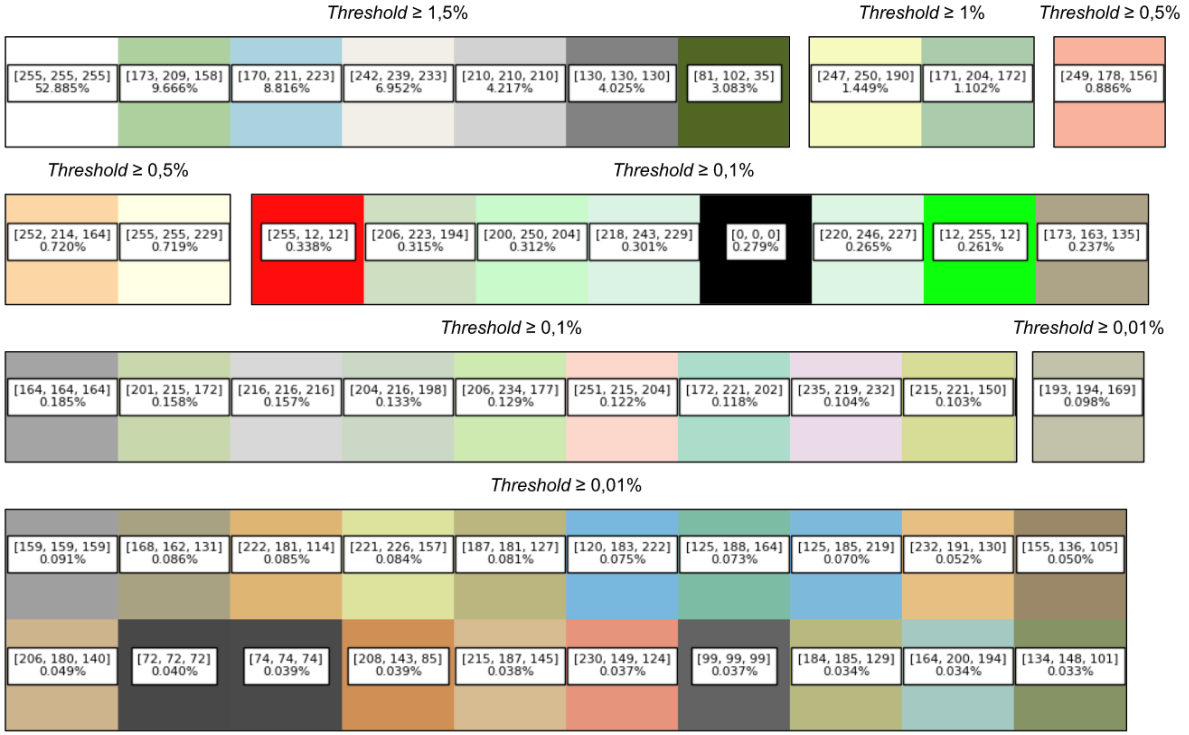
App “Meteorito”	
Threshold	

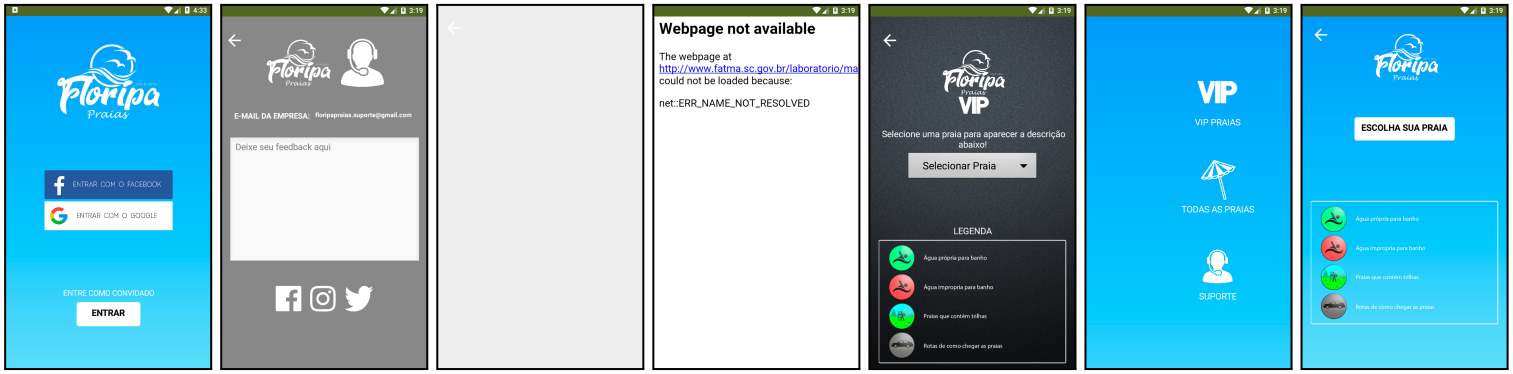
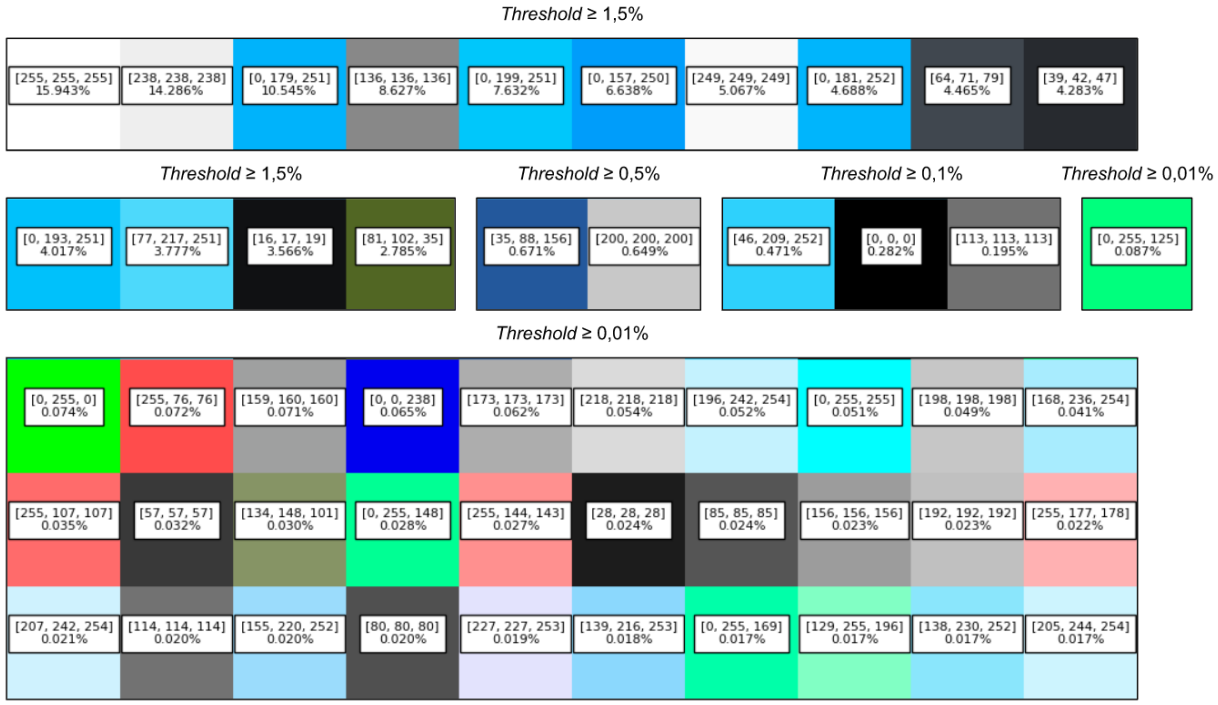
App “IMC”	
Threshold	*Threshold* ≥ 1,5% [244, 218, 203] 13.097% · [255, 255, 255] 12.105% · [230, 197, 164] 8.496% · [144, 100, 73] 5.832% · [130, 130, 130] 4.287% · [218, 217, 223] 4.206% · [36, 39, 58] 3.981% · [174, 130, 101] 3.630% · [81, 102, 35] 3.398% · [254, 236, 198] 3.142% *Threshold* ≥ 1,5% [63, 68, 98] 2.869% · [0, 0, 0] 2.801% · [251, 92, 73] 2.711% · [244, 63, 44] 2.234% · [215, 173, 131] 1.991% · [96, 114, 152] 1.937% *Threshold* ≥ 1% [200, 181, 177] 1.476% · [166, 198, 122] 1.131% · [183, 153, 125] 1.017% · [208, 51, 36] 1.014% *Threshold* ≥ 0,5% [250, 111, 92] 0.778% · [191, 192, 222] 0.755% · [146, 165, 75] 0.723% · [19, 21, 33] 0.668% · [137, 167, 97] 0.534% *Threshold* ≥ 0,1% [104, 78, 61] 0.428% · [170, 42, 29] 0.420% · [70, 70, 70] 0.381% · [125, 141, 174] 0.372% · [134, 148, 101] 0.363% *Threshold* ≥ 0,1% [252, 135, 118] 0.357% · [165, 165, 165] 0.324% · [105, 143, 40] 0.318% · [68, 87, 129] 0.315% · [219, 196, 212] 0.311% · [41, 25, 12] 0.293% · [107, 128, 181] 0.289% · [194, 185, 108] 0.281% · [202, 75, 60] 0.260% · [103, 103, 103] 0.241% [108, 125, 69] 0.237% · [205, 183, 125] 0.232% · [145, 144, 79] 0.231% · [47, 52, 20] 0.231% · [154, 175, 48] 0.218% · [131, 156, 210] 0.214% · [254, 151, 108] 0.212% · [141, 185, 88] 0.209% · [186, 210, 116] 0.208% · [195, 200, 80] 0.206%

App “Medike”	
Threshold	

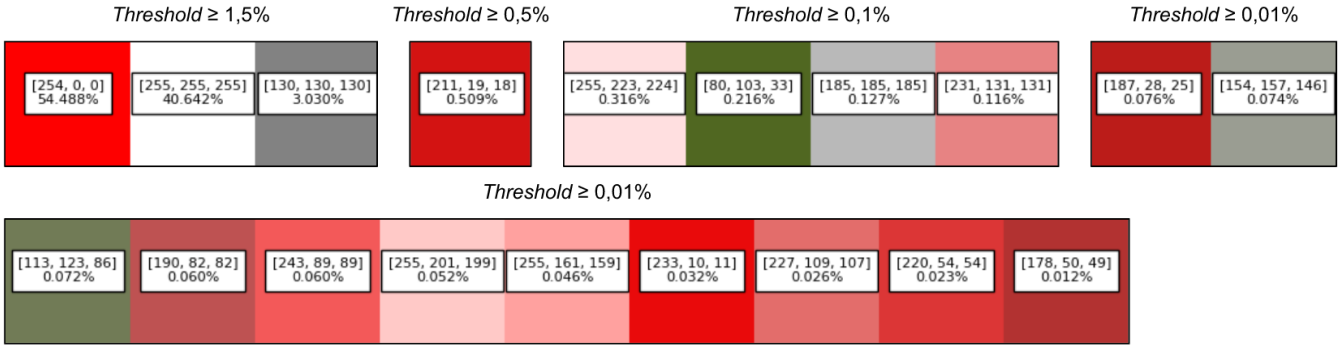
App “QArvore”	
Threshold	

App “WeLearn”	
Threshold	

App “vaiDeBike”	
Threshold	

App “FloripaPraias”	
Threshold	

App “Pikachu”	
Threshold	*Threshold* ≥ 1,5% [250, 250, 250] 54.317% [249, 221, 77] 14.109% [255, 255, 120] 7.445% [255, 200, 0] 3.974% [130, 130, 130] 3.582% [81, 102, 35] 3.084% [197, 197, 197] 2.521% [222, 193, 64] 2.444% [255, 243, 129] 1.760% *Threshold* ≥ 0,5% [218, 189, 87] 0.659% *Threshold* ≥ 0,5% [195, 158, 51] 0.642% [252, 231, 153] 0.598% [248, 90, 58] 0.545% [255, 219, 105] 0.532% *Threshold* ≥ 0,1% [61, 40, 28] 0.463% [37, 32, 36] 0.398% [251, 241, 203] 0.381% [255, 255, 0] 0.364% [133, 83, 36] 0.242% [197, 155, 76] 0.241% *Threshold* ≥ 0,1% [168, 127, 44] 0.157% [255, 115, 90] 0.108% [255, 222, 0] 0.105% [76, 54, 74] 0.103% *Threshold* ≥ 0,01% [231, 85, 69] 0.098% [255, 207, 77] 0.085% [218, 190, 114] 0.075% [211, 190, 138] 0.072% [192, 147, 95] 0.065% [208, 75, 41] 0.062% *Threshold* ≥ 0,01% [255, 217, 127] 0.061% [134, 56, 31] 0.054% [222, 226, 213] 0.047% [222, 169, 52] 0.043% [60, 52, 50] 0.035% [255, 253, 157] 0.031% [134, 148, 101] 0.031% [208, 182, 151] 0.031% [87, 50, 31] 0.026% [255, 251, 185] 0.026% [129, 121, 66] 0.022% [250, 138, 73] 0.021% [255, 198, 110] 0.018% [104, 104, 104] 0.018% [168, 140, 119] 0.017% [109, 79, 100] 0.017% [227, 211, 171] 0.017% [91, 81, 76] 0.016% [106, 97, 57] 0.016% [247, 160, 60] 0.015%

App “sos”	
Threshold	

ANEXO D

O código da implementação da solução pode ser encontrado no repositório do gitlab, acessado pelo *link*: https://codigos.ufsc.br/gqs/app-inventor-palette-originality.git.

Desenvolvimento de um Modelo para Avaliação da Originalidade da Paleta de Cores no Design de Interface de Aplicativos Android

Igor Perazzoli[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

igor.perazzoli@grad.ufsc.br

Abstract. *The objective of this researchis to develop and test a model capable of evaluating the originality of the color palette in the interface design of Android applications developed with App Inventor. The model helps the student's learning process, providing an automated feedback on the originality of the application's color palette created by the student.*

Resumo. *O objetivo deste artigo é desenvolver e testar um modelo capaz de avaliar a originalidade da paleta de cores no design de interface de apps Android desenvolvidos com App Inventor. O modelo auxilia o processo de aprendizagem do aluno, fornecendo uma avaliação de feedback da originalidade da paleta de cores do aplicativo criado por ele de forma automatizada.*

1. Introdução

Segundo a Base Nacional Comum Curricular (BNCC) o ensino da computação na escola consegue ser uma ferramenta que também permite estimular a criatividade (Brasil, 2022). O App Inventor é um ambiente de programação visual intuitiva que permite construir aplicativos funcionais para sistemas Android.No contexto de estimular a criatividade no desenvolvimento de apps espera-se também o estímulo na criação de design de interface criativos e originais. Sendo que um design de interface é considerado original em termos do design visual, especialmente em relação à paleta de cores. Assim, uma interface de usuário pode ser considerada original em termos da paleta de cores utilizada se ela é diferente das paletas tipicamente utilizadas (Alves et al., 2020).

Para auxiliar o aluno na aprendizagem da criatividade se faz necessária uma forma de retornar um feedback do desempenho do aluno (Mishra e Henriksen, 2013). A avaliação da originalidade da paleta de cores dos aplicativos é complexa. Ao procurar por pesquisas relacionadas constata-se que ainda não há soluções para a avaliação da originalidade da paleta de cores em aplicativos, somente em interfaces web e de desktop.

Dessa forma, o modelo a ser construído neste trabalho pretende facilitar a avaliação dos aplicativos de alunos por profissionais de educação, que, por vezes, não têm o conhecimento necessário para julgar o quanto uma paleta de cores de uma interface é original, além de reduzir o seu esforço e tempo nessa tarefa.

2. Estado da Arte

O objetivo do mapeamento feito no estado da arte é responder quais abordagens existem para automaticamente analisar a originalidade de paletas de cores de aplicativos (com App Inventor).

Os artigos são buscados nos principais bancos de dados e bibliotecas digitais no campo da computação, incluindo as Bibliotecas Digitais ACM, a IEEE Xplore e Scopus com acesso por meio do Portal Capes. A pesquisa também foi feita via Google Scholar, para complementar a busca (Haddaway et. al., 2015, Piasecki et al. 2017).

A partir do resultado inicial das buscas, foram selecionados artigos potencialmente relevantes de acordo com os critérios de inclusão e exclusão por meio de uma análise do título, resumo e palavras-chave de cada artigo. Como resultado final foram identificados 3 artigos relevantes.

Dentre as abordagens relevantes analisadas, é possível observar que nenhuma está atrelada ao contexto educacional, sendo todas contextualizadas no ambiente profissional. Outra observação importante nesse primeiro momento é em relação às interfaces utilizadas, em que nenhum dos três artigos apresenta uma solução para aplicativos Android e, portanto, não há ocorrências envolvendo o uso do App Inventor.

Os artigos possuem distinções muito claras entre suas propostas, e uma característica discrepante entre as três propostas é o uso de modelos de cores variadas. Um modelo utiliza o sistema RGB, outro utiliza o CIE L*c*h (uma variante do CIE L*a*b) e outro utiliza o HSV. Dessa forma observa-se que parece não ter um consenso sobre o modelo de cores a ser utilizado neste tipo de contexto.

Quanto às técnicas de análise adotadas pelas abordagens é notável que também não há consenso. É possível observar que algumas métricas estão diretamente associadas ao sistema de cores utilizado na abordagem. Enquanto há outras que utilizam métricas desenvolvidas especialmente para a situação que deseja analisar.

De forma geral, observa-se que atualmente as abordagens voltadas à avaliação da originalidade de paletas de cores de apps são quase inexistentes. Foram encontradas somente três abordagens voltadas aos sistemas de software de forma geral, porém não foi encontrado nenhum especificamente voltado a apps, menos ainda a apps desenvolvidos com App Inventor.

3. Proposta da Solução

A proposta da solução busca desenvolver um modelo que avalia a originalidade da paleta de cores de um aplicativo feito com App Inventor em comparação com um universo de referência composto por outros aplicativos também feitos com App Inventor em contextos educacionais semelhantes.

3.1. Análise dos requisitos

Para avaliar a paleta de cores é utilizado um conjunto de 1.773 diretórios, sendo que cada diretório contém as capturas de todas as telas de um determinado aplicativo (as imagens das capturas de tela estão no formato “png” com tamanho igual a 1.080 pixels de largura por 1.920 pixels de altura), totalizando 9.050 telas.

O usuário deve inserir o caminho de origem de uma pasta contendo capturas de todas as telas do aplicativo que deseja ter a originalidade de sua paleta de cores avaliada. A abordagem desenvolvida visa avaliar o grau da originalidade referente a paleta de cores de apps feitos com App Inventor. Para isso é definida uma escala do grau de

originalidade da paleta de cores que varia de 0 (paleta menos original) até 10 (paleta mais original) ao comparar a paleta com o universo de referência.

3.2. Modelagem

A abordagem consiste em extrair as cores das capturas de telas do aplicativo, filtrar as cores mais relevantes gerando a paleta de cores proporcionais do aplicativo, comparar essa paleta com as paletas do universo de referência, atribuir um valor de originalidade em relação com cada paleta, para enfim avaliar o grau de originalidade do aplicativo (Figura 1).

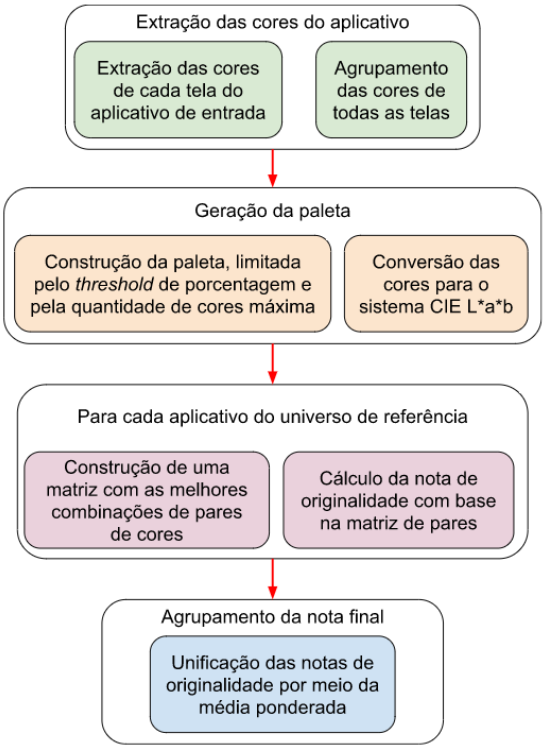


Figura 1. Ordem de execução das partes da modelagem

Como parte da solução proposta, a configuração da entrada esperada é dada por um conjunto de capturas de todas as telas do aplicativo que deve ser avaliado.

A extração das cores do aplicativo é feita por meio das capturas de tela geradas pelo aplicativo. Para extrair as cores é utilizada a biblioteca “extcolors” (Pypi, 2020) da linguagem Python que extrai as cores a partir de uma imagem.

3.2.1. Estudo comparativo da originalidade das cores entre aplicativos

Com o objetivo de ter uma base para definir o cálculo das notas da solução desenvolvida é relevante produzir um estudo que considere a percepção humana real (questionando pessoas sobre a originalidade da paleta de cores em relação a um universo de referência pequeno).

O estudo foi realizado por meio de uma pesquisa no período de março a junho de 2022. Ao final foram contabilizadas as respostas de um total de 96 participantes convidados via redes sociais e emails, selecionados por conveniência. Sendo que a

maioria (60 participantes) não possui experiência com design visual e/ou design de interfaces, 59 participantes estão na faixa etária de 18 a 29 anos e 6 responderam que possuem algum grau de daltonismo.

Considerando a avaliação das capturas de tela como a avaliação mais fiel à percepção humana para avaliar a similaridade entre os aplicativos (padrão ouro), é possível extrair as médias obtidas para cada aplicativo em cada cenário e compará-las entre si (Tabela 1).

Tabela 1. Comparação entre as médias de similaridade de cada aplicativo em comparação ao aplicativo base

App	Média das originalidades		
	Capturas de tela	Paleta em blocos	Paleta proporcional
suChef	3.2	4.6	4.6
Chazaum	3.3	4.7	4.6
Medike	3.7	4.6	3.2
QArvore	4.3	4.8	4.9
Pikachu	4.9	3.0	2.6
AcheiOseuEmprego	6.1	5.1	5.7
InfoCarvalhal	6.5	4.6	5.4
RotasSeguras	6.9	6.4	6.9
IMC	7.5	8.3	8.1
MoveWithMe	8.6	8.9	9.1

Após avaliar as notas obtidas utilizando a correlação de Spearman, conclui-se que é possível representar as cores de um aplicativo usando a representação por paletas, e que a representação de paleta de cores proporcional se aproxima mais da percepção humana da paleta de cores em relação às interfaces de apps.

3.2.2. Definição do modelo de cores

Ao analisar os sistemas de cores utilizados nas pesquisas levantadas no estado da arte é possível obter a interpretação de que não há um consenso definido sobre qual seria o melhor sistema de cores para representar a paleta de cores de um aplicativo. Considerando isso, o sistema de cores escolhido para fazer isso é o CIE LAB (Ibraheem et al., 2012). A opção por esse sistema se deve ao fato de ser capaz de se assemelhar muito com a forma que o ser humano capta as cores pela visão, sendo possível julgar mais apropriadamente o nível de similaridade entre as cores (Ibraheem et al., 2012).

3.3. Definição dos parâmetros da paleta de cores

Ao extrair as cores de um aplicativo é possível perceber que o total de cores obtidas é muito maior do que a percepção humana consegue compreender. Isso é notável principalmente quando há imagens ou efeitos de degradê no aplicativo.

Para resolver esse problema de excesso de cores, é necessário mesclar cores muito semelhantes com o objetivo de reduzir o número de cores obtidas na extração.

Para realizar essa mesclagem das cores é utilizado o parâmetro de tolerância contido na função de extração “extract_from_path()” da biblioteca “extcolors” (Pypi, 2020). É por meio dele que se limita o quão similares as cores devem ser, para que ambas sejam unidas em apenas uma cor.

A biblioteca utilizada define como padrão o valor 32 para a tolerância. No entanto, ao incluir valores inferiores à 32, mas ainda elevados (como 20 e 25), é notável que há tons de cores possíveis de serem diferenciados pela percepção humana que são mesclados em uma única cor.

Contudo, utilizar um valor de tolerância muito baixo para telas com quantidades de cores maiores tende a levar muito mais tempo para realizar a extração. Portanto, para definir o valor de tolerância é necessário encontrar um equilíbrio entre o tempo de extração e o nível tolerável de mesclagem das cores. Consequentemente, o valor que melhor atende estes requisitos é o 10, pois mescla cores suficientemente próximas em um tempo razoável considerando situações extremas.

O threshold de porcentagem é um parâmetro utilizado para delimitar a porcentagem mínima que uma cor deve apresentar para ser incluída na paleta de cores do aplicativo. Esse valor é definido principalmente pela relevância das cores selecionadas, tomando o cuidado para não deixar o valor tão baixo ao ponto de tornar o parâmetro quase irrelevante.

Usar valores como 1,5%, 1% e 0,5% reduzem consideravelmente a quantidade de cores que devem aparecer na paleta final. Contudo, a perda por usar esses valores pode ser grande em algumas situações, eliminando cores que aparecem pouco no aplicativo, mas que são importantes para compor a paleta.

Há determinados casos em que somente um threshold de 0,01% seria capaz de captar algumas cores perceptíveis pela visão humana. Contudo, permitir essa flexibilidade faz com que cores demais sejam incluídas na paleta, tornando o parâmetro inviável e os cálculos de originalidade mais custosos. Por isso, o valor escolhido foi 0,1% por ser o melhor equilíbrio entre essas situações.

Outro parâmetro importante é o que limita a quantidade de cores máxima que uma paleta pode possuir. Esse parâmetro é fundamental para o cálculo da similaridade, pois por meio dele é possível reduzir o tempo de execução.

Considerando o tempo de execução é possível concluir que a melhor opção seria utilizar um valor limite da quantidade de cores muito pequeno, reduzindo drasticamente o tempo total de execução da solução. Contudo, usar um valor de limite muito pequeno descartaria uma quantidade de cores significativas da paleta do aplicativo. Por isso, o valor limite da quantidade de cores escolhido é 20, pois o tempo de execução pode ser tolerável (dependendo da quantidade de aplicativos contidos no universo de referência) sem cortar cores relevantes na maioria dos casos.

Com isso, fica definido que a paleta de cores definitiva de um aplicativo deve ser extraída usando valor de tolerância 10, valor limite de threshold de porcentagem igual a 0,1% e a quantidade de cores presentes na paleta limitada a 20 cores no máximo.

3.4. Cálculo da originalidade entre dois aplicativos

O cálculo da originalidade entre dois aplicativos é usado para atribuir uma nota de originalidade da paleta de cores ao aplicativo dado como entrada em relação a outro selecionado.

Em termos conceituais, a paleta do aplicativo dado como entrada é comparada individualmente com as paletas de cada aplicativo contido no universo de referência. Ao selecionar um aplicativo do universo de referência, as cores de ambos aplicativos são “pareadas”, isto é, cada cor do aplicativo de entrada é colocada em um par com a respectiva cor mais similar do aplicativo selecionado. Assim, a nota da originalidade da paleta de cores é resultado do somatório de cálculos de similaridade feitos para cada par de cores gerados.

Inicialmente, uma função recebe duas paletas de dois aplicativos distintos (aplicativo de entrada e um aplicativo selecionado do universo de referência) e gera uma matriz de similaridade, de forma que as linhas representam as cores de uma paleta, as colunas representam as cores da outra paleta e as células são preenchidas com o valor de similaridade (calculado pela função “delta_E” da biblioteca “colour”) das duas cores contidas naquele índice.

Em seguida, a matriz de similaridade passa pelo algoritmo húngaro para definir as melhores combinações de cores possíveis para aquelas paletas. Ao selecionar uma cor da primeira paleta, a função indica qual cor da segunda paleta possui a maior similaridade e une ambas em um par. Contudo, se a função encontra uma cor da primeira paleta ainda mais similar com a cor da segunda paleta atribuída a um par anteriormente, um novo par é formado com essas duas cores e a outra cor da primeira paleta, que havia ficado sem par, recebe a segunda cor mais similar com ela (Figura 2).

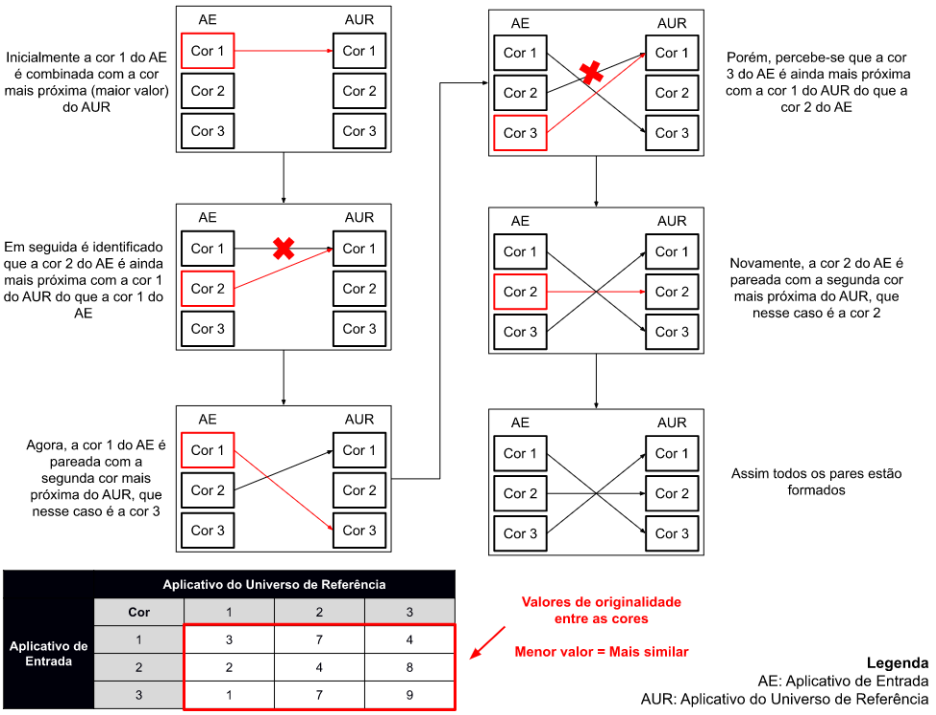


Figura 2. Passos da execução do algoritmo húngaro

Com os pares formados, o cálculo da nota de originalidade da paleta pode ser efetivamente feito (Figura 3).

$$nota = \sum\{[(|pctg_1 - pctg_2| \times peso_{pctg} + \Delta E2000(cor_1, cor_2) \times peso_{\Delta E}] \times pctg_1 \times ajuste\}$$

Figura 3. Cálculo da nota de originalidade da paleta de cores de um aplicativo

A nota é resultado do somatório de "notas individuais" dadas para cada par de cores definidos anteriormente, sendo que cada uma dessas "notas" terá seu valor ajustado tanto pela porcentagem que a cor do aplicativo principal ocupa "pctg1" e por um fator de "ajuste" que permite formatar a nota para o formato mais adequado (nesse caso o valor utilizado é 0,1 para que a nota esteja entre 0 e 10).

A "nota individual" de cada cor é gerada pela soma da diferença entre as porcentagens das duas cores e do valor de similaridade calculado pelo "delta_E" já contido na matriz. No entanto, esses dois valores precisam de um ajuste para dar uma nota mais realista ao julgamento humano.

3.5. Cálculo da nota final de originalidade

Com as notas de originalidade geradas a partir da comparação com cada paleta do universo de referência, é necessário utilizar algum meio de agrupá-las em uma única nota (Figura 4). A maneira mais eficaz encontrada de fazer esse agrupamento é por meio da média ponderada. Para calcular essa média ponderada é preciso atribuir diferentes pesos para as notas obtidas. Isso é feito com a ajuda de uma média simples de todas as notas (em que todas as notas possuem o mesmo peso).

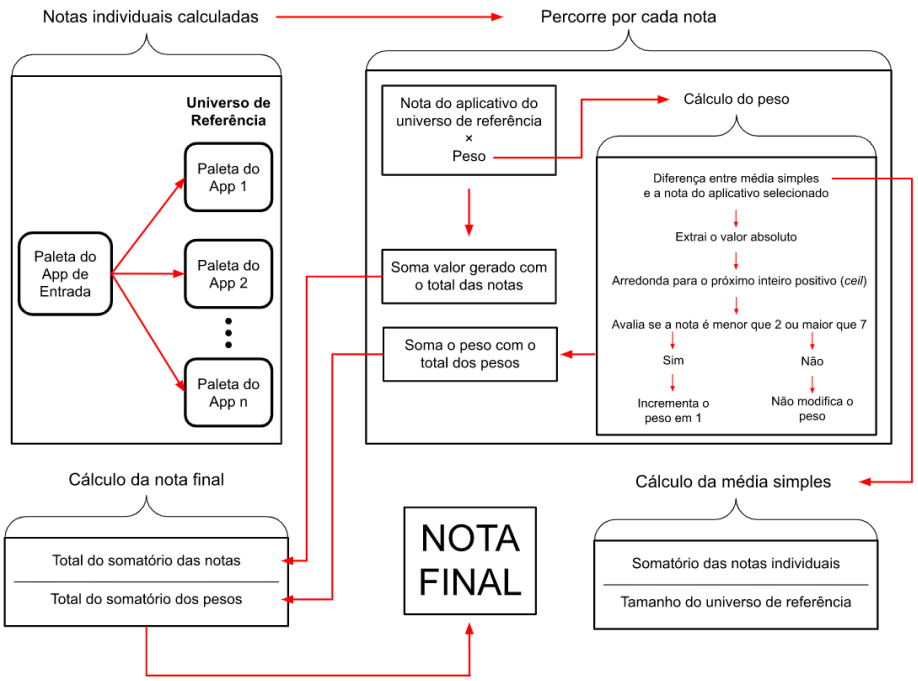


Figura 4. Representação do cálculo da nota final da originalidade

O cálculo do “peso atual” (peso da nota atual do aplicativo do universo de referência comparado) é feito ao obter o valor absoluto da diferença entre a “nota atual” com o valor da média simples. De forma conceitual, quanto mais distante uma nota estiver da média simples, maior será seu peso para o cálculo da nota final. A escolha do peso de ser calculado dessa forma vem da ideia de que notas mais distantes, sejam notas próximas a 10 ou próximas a 0, devem ter um impacto maior por representarem comparações entre paletas muito ou pouco originais entre si.

Seguindo a ideia de que notas mais distantes têm maior impacto na nota final, existem duas situações especiais em que o peso é ainda maior (acrescenta mais 1 ao “pesoatual”), notas menores que 2 ou maiores que 7. Esses casos foram definidos pois são as notas mais importantes contidas dentro de todo o conjunto. Inicialmente os valores deveriam ser equivalentes (2,5 para o limite inferior e 7,5 para o limite superior), porém essa proporção foi desequilibrada para flexibilizar um pouco mais as notas dentro do contexto da Educação Básica.

3.5. Avaliação do cálculo da nota final de originalidade

Com base na comparação observa-se que a solução proposta pode ser considerada precisa, pelo motivo da nota de originalidade obtida pelo aplicativo de exemplo se aproximar suficientemente da nota alocada pelas pessoas que responderam ao estudo da percepção humana.

Em termos do tempo de execução, o cálculo leva aproximadamente 1 minuto e 35 segundos para retornar a nota final de originalidade do aplicativo base utilizado ao comparar com os aplicativos do universo de referência, sendo 19 segundos para gerar a paleta do aplicativo dado como entrada e 76 segundos para calcular a originalidade. Tempos inferiores poderiam ser obtidos ao reduzir o limite da quantidade de cores presentes nas paletas, porém isso pode influenciar negativamente a nota final por ter uma quantidade menor de cores sendo comparadas.

Assim, assume-se que o tempo de execução é considerado razoável, mesmo não sendo tão próximo de um tempo “instantâneo”, ideal para o contexto da Educação Básica. Na prática, é possível considerar que se trata de um tempo longo demais. Uma alternativa para aprimorar esse tempo seria calibrar melhor o universo de referência, observando o comportamento dos aplicativos em diferentes avaliações para descartar aqueles que não agregam nenhum valor para o cálculo da nota, reduzindo ele para um tamanho mínimo, porém ainda representativo.

4. Implementação

A implementação da solução é feita por meio da linguagem de programação “Python 3.10” (https://www.python.org/). Uma linguagem escolhida pela simplicidade e familiaridade com a linguagem, além de haver bibliotecas importantes para o desenvolvimento.

A arquitetura da solução (Figura 5) é separada em 4 arquivos, sendo 3 classes e 1 arquivo contendo funções adicionais usadas pelas classes. Sendo que o código fonte da implementação está disponível em um repositório do gitlab (https://codigos.ufsc.br/gqs/app-inventor-palette-originality.git).

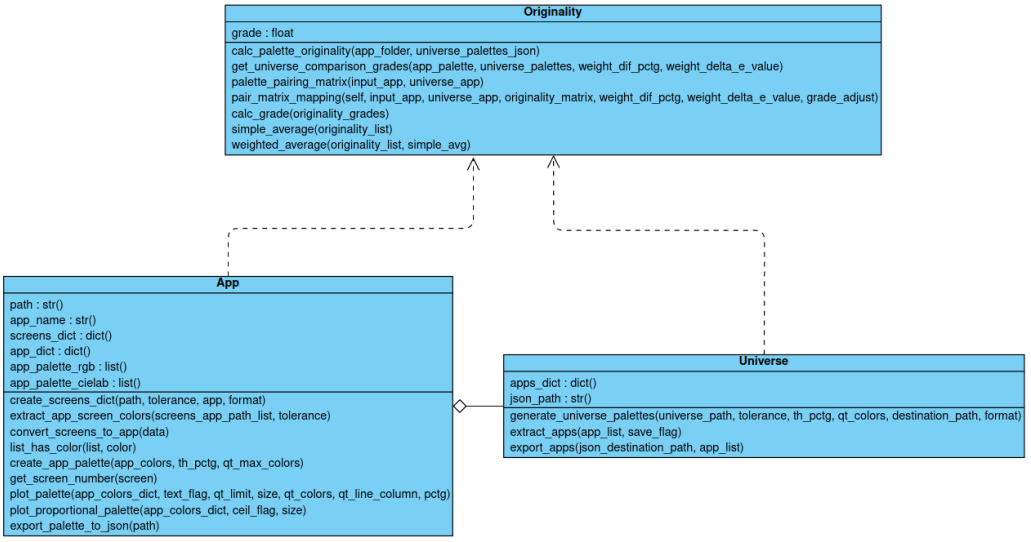


Figura 5. Diagrama de classes da implementação

5. Conclusão

Por meio desse trabalho se propõe uma solução para a avaliação da originalidade da paleta de cores em aplicativos Android dentro do contexto de Educação Básica, um tema ainda pouco abordado atualmente.

Por meio de um mapeamento sistemático, foram levantados poucos trabalhos no estado da arte. Sendo que os três trabalhos obtidos ao final da revisão apresentaram pouco (ou em alguns casos nenhum) consenso para auxiliar nas decisões tomadas sobre os modelos de avaliação da originalidade das cores. No entanto, foi possível extrair algumas informações sobre os modelos de cores utilizados e sobre algumas formas de verificar a similaridade entre as cores.

Um modelo de avaliação da originalidade da paleta de cores em aplicativos foi desenvolvido, usando métodos de extração e análise que minimizam o impacto de trabalhar com uma quantidade de cores reduzida, sendo que a nota atribuída permanece coerente com a percepção humana. A maior dificuldade do trabalho estava em encontrar o ponto mais próximo possível da percepção humana para avaliar a originalidade da paleta de cores de um aplicativo. Na tentativa de alcançar a melhor aproximação diferentes fatores foram considerados (uso do sistema de cores CIE L*a*b, estudo da percepção humana para as paletas, considerar a porcentagem das cores na tela, parâmetros de ajuste e limite para cálculos, entre outros). Procurando minimizar os riscos de ameaças à validade, cada fator recebeu uma atenção especial para definir qual seria a abordagem mais adequada a ser adotada. O sistema CIE L*a*b foi escolhido por ser o sistema mais coerente com a percepção humana de cores que há atualmente. O estudo da percepção humana utilizou valores iniciais (de quantidade de cores e de porcentagem presente no aplicativo) pensados para facilitar a forma com que o ser humano julga as paletas. E os parâmetros para a avaliação entre um par de aplicativos foram especificados ao analisar um conjunto de exemplares utilizando diferentes valores.

Assim, os resultados obtidos na avaliação da solução proposta em comparação com as notas manualmente alocadas por humanos apontam resultados positivos. Aplicando a mesma metodologia para ambos os casos (notas manuais e notas atribuídas automaticamente pela solução), a diferença no resultado final é de apenas 0,5 ponto a mais para as notas atribuídas pela solução, representando uma diferença aceitável, considerando o contexto de Educação Básica em que o tema está inserido.

Outra vantagem em utilizar o modelo proposto está no tempo total de execução. Mesmo ao executar a avaliação da originalidade da paleta do aplicativo base para um universo de referência contendo 1773 aplicativos, o resultado foi gerado em aproximadamente 1 minuto e 30 segundos de execução, um valor inferior ao tempo normalmente levado por um avaliador humano para gerar a nota.

Dessa forma espera-se que o modelo desenvolvido neste trabalho possa contribuir para apoiar a avaliação da originalidade de aplicativos desenvolvidos por estudantes da educação básica.

Trabalhos futuros incluem a implementação do módulo dentro da ferramenta Codemaster (Alves, 2019) para possibilitar o acesso online a esse tipo de apoio de avaliação. Também deverá ser feito um estudo de caso, aplicando o modelo proposto na prática, avaliando as paletas de aplicativos desenvolvidos por uma turma na educação básica.

Referências

Alves, N. d. C., da., Gresse Von Wangenheim, C., Alberto, M., Martins-Pacheco, L. H. Uma Proposta de Avaliação da Originalidade do Produto no Ensino de Algoritmos e Programação na Educação Básica. In: Anais do Simpósio Brasileiro de Informática na Educação, Natal, Brasil, 2020.

Haddaway, N. R., Woodcock, P., Macura, B., Collins, A. Making literature reviews more reliable through application of lessons from systematic reviews. Conservation Biology, v. 29, issue 6, p. 1596-1605, 2015. https://doi.org/10.1111/cobi.12541

Ibraheem, N., Hasan, M., Khan, R. Z. Understanding Color Models: A Review. ARPN Journal of Science and Technology, 2012.

Mishra, P., Henriksen, D. A NEW Approach to Defining and Measuring Creativity: Rethinking Technology & Creativity in the 21st Century. TECH TRENDS, 57, 10-13, 2013. https://doi.org/10.1007/s11528-013-0685-6

Piasecki, J., Waligora, M., Dranseika, V. Google Search as an Additional Source in Systematic Reviews. Sci Eng Ethics 24, p. 809-810, 2018. https://doi.org/10.1007/s11948-017-0010-4

Pypi. Extcolors 1.0.0. 2020. Disponível em: https://pypi.org/project/extcolors/. Acesso em: 10 de outubro de 2022