Lift Management - Part 1

AIAD 18/19 - Grupo 50

António Almeida (up201505836)
Diogo Torres (up201506428)
João Damas (up201504088)

Descrição do problema

- Sistema para gestão de uso de um conjunto de elevadores num edifício, com o intuito de tornar a utilização dos mesmos eficiente, reduzindo o tempo de espera de cada pedido.
- Os elevadores são representados por agentes que têm uma capacidade limite, bem como um tempo de transporte entre pisos fixo.
- O edifício também é representado por um agente que tem como propósito gerar pedidos e desencadear o(s) protocolo(s) necessário(s) para a atribuição de tarefas (pedidos) a um elevador
- Os pedidos a elevadores são acontecimentos estocásticos e a sua ocorrência pode ser parametrizada (para refletir, por exemplo, diferentes padrões de pedidos característicos de diferentes alturas de um dia).
- Existência de diferentes cenários, nomeadamente o botão de chamada em cada piso apenas indicar a direção desejada (subir/descer) *vs* indicar o piso de destino

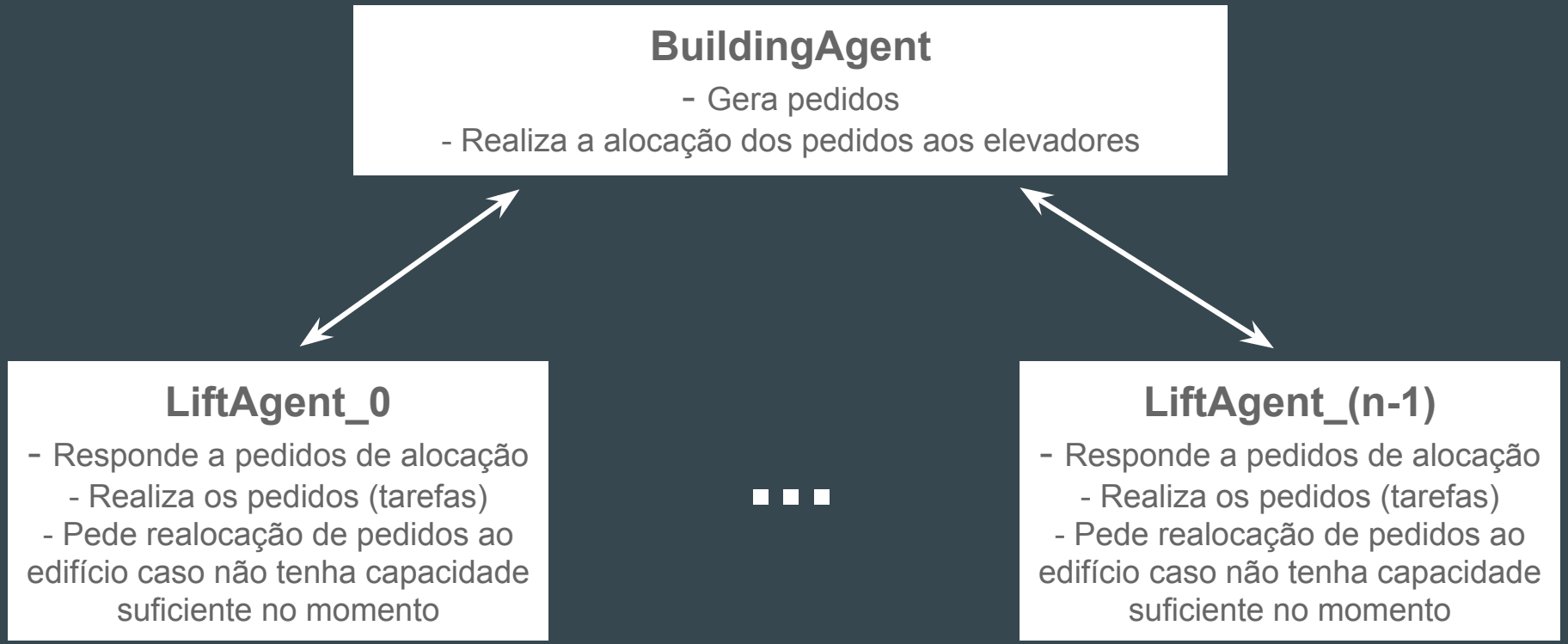
Variáveis independentes:

- Número de pisos no edifício
- Número de elevadores
- Capacidade máxima de cada elevador
- Tempo de transporte entre pisos
- Probabilidade/Frequência de chamadas em cada piso
- Tipo de chamada inicial ao elevador (subir/descer ou piso destino conhecido a priori)

Variáveis dependentes:

- Taxa ocupação por elevador
- Taxa uso (tarefas executadas) por elevador
- Tempo espera mínimo,máximo de pedidos (por elevador e global)

Esquema global



Interação e protocolos (comunicação)

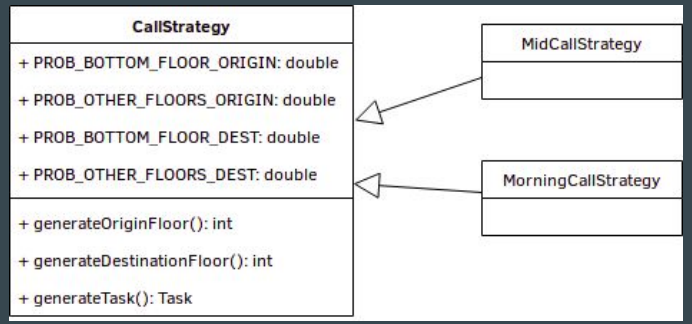
- **FIPA Contract Net**
  - BuildingAgent: a cada período da frequência de novas chamadas parametrizado, envia uma mensagem para todos os elevadores contendo o pedido a servir (*prepareCfps*); avalia os scores devolvidos pelos elevadores (*handleAllResponses)*, enviando ACCEPT ao elevador com menor score (menor "trabalho" consoante o critério) e REFUSE aos restantes; recebe INFORM do elevador respetivo a confirmar receção - classe *CallGenerator*
  - LiftAgent: recebe um pedido e calcula o score consoante a estratégia de avaliação parametrizada (*handleCfps)*; recebendo um ACCEPT, adiciona ao seu conjunto de tarefas (*handleAcceptProposal)* - classe *CallResponder*

- **FIPA Request**
  - LiftAgent: não conseguindo realizar um pedido por falta de capacidade, pede ao edifício para realocar parte do mesmo (*prepareRequests)*; ao receber aceitação por parte do edifício, sabe que o pedido será efetivamente realocado - classe *CallReRequester*
  - BuildingAgent: recebendo um pedido de um elevador que não pode ser cumprido naquele momento pelo mesmo (*handleRequest*), adiciona-o a um buffer local, de onde o retirará no próximo período de frequência de chamada - classe *Recaller*

Arquiteturas dos agentes e estratégias utilizadas (I)

Building Agent

- **callStrategy -** Objeto responsável por gerar tarefas (pedidos). Concretizado em 2 classes representando diferentes padrões de pedidos (apenas diferem na inicialização dos valores das probabilidades) - Morning foca pedidos no piso 0, Mid nos restantes.
- **floors -** Vetor com as representações de cada andar. Antes de iniciar o protocolo de atribuição (Contract Net), "ativa" o piso de origem de modo a representar visualmente a origem e sentido do destino do próximo pedido durante o período do movimento dos elevadores de 1 andar.
- **newTasks -** Ao receber pedidos de volta dos elevadores, estes são guardados num vetor e têm prioridade em relação a novos pedidos aquando a execução do protocolo de alocação.




```java
//BuildingAgent.java
private void newCall(Task task) {
    //Create ACLMessage...
    floors.get(task.getOriginFloor()).activate(task.getDirection());
    //Add requester behaviour
}
```

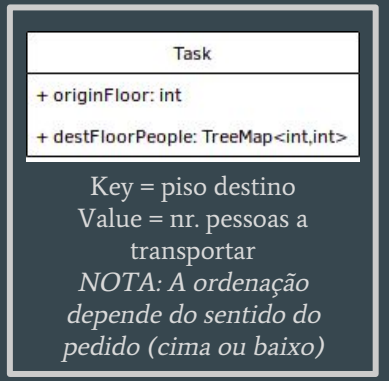
```java
//Floor.java
public void activate(Direction direction) {
    this.direction = direction;
    ++counter;
}
@Override
public void draw(SimGraphics simGraphics) {
    if (counter > 0) {
        simGraphics.drawImageToFit(direction == Direction.DOWN ? Floor.down : Floor.up);
        counter = (counter + 1) % lift_speed;
    } else
        simGraphics.drawRect(Color.RED);
}
```

```java
public void newCall() {
    if (newTasks.isEmpty())
        newCall(callStrategy.generateTask());
    else
        newCall(newTasks.remove( index: 0));
}
```

(De notar que newCall() é o método que está *scheduled* para ser chamado periodica e automaticamente na classe *Launcher*)

Arquiteturas dos agentes e estratégias utilizadas (II)

Lift Agent



- **tasks** - Vetor com os pedidos pendentes alocados ao elevador. É auxiliado pelos atributos **currentTask** (pedido atual a ser processado), **goingToOrigin** (indica se, no pedido atual, o elevador se desloca para a origem ou destino(s)) e **state** (direção de movimento atual; utilizado para atualizar corretamente a posição)
- **addTask(Task)** - Aquando a receção de um pedido (ACCEPT_PROPOSAL), este é preferencialmente agregado a outro pedido em **tasks** semelhante (mesma origem e destino em sentido semelhante) em vez de adicionado de forma independente, de forma a potencialmente minimizar a espera
- **executeTasks()** - Função que, a cada período de movimento, é responsável por verificar se o agente tem pedidos para realizar e desencadear a execução do próximo pedido, se for o caso:
  - **startTask()** - O elevador desloca-se até à origem do pedido, onde começa efetivamente o processamento do mesmo através desta função. O algoritmo utilizado consiste em:
    i. Verificar se o elevador possui capacidade para o número total de pessoas total no pedido
       - Se sim, todas são recolhidas
       - Se não, é escolhido um número aleatório de pessoas de um determinado piso para encher o elevador, repetindo este processo até o elevador estar cheio.

    ii. Caso o elevador não consiga transportar todas as pessoas, reenvia ao edifício um *request* para realocar o novo pedido com as restantes pessoas.
  - **endTask()** - Ao deixar as pessoas no piso de destino, o elevador ou parte para o próximo destino (no caso de ter pedidos agregados), ou então pára à espera de novos pedidos.

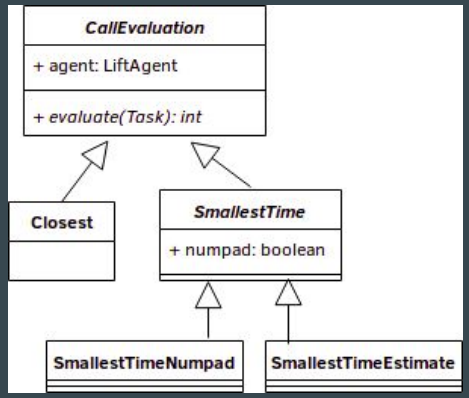
Arquiteturas dos agentes e estratégias utilizadas (III)

Lift Agent

- **evaluator -** Objeto responsável por avaliar tarefas (pedidos). Concretizado em 3 classes representando diferentes padrões de avaliação:
  - *Closest*: Score = abs(elevador.currentFloor - pedido.originFloor)
  - *SmallestTime*: Concretizado em 2 classes. Aqui, o score é dado pelo cálculo do tempo que o elevador demoraria a servir os pedidos que tem pendentes antes de poder servir este. Para tal, cada pedido na lista de tarefas a executar no momento é avaliado individualmente tendo em conta o estado atual do elevador e o score atualizado de forma adequada:
    - Caso o pedido corresponda à **currentTask**, é somado o tempo de transporte até ao destino deste. Para além disso, é acrescentado o tempo até à chegada à origem, caso ainda não tenha lá chegado (**goingToOrigin** == true)
    - Caso contrário, é avaliado se seria possível agregar os pedidos. Se sim, apenas é somado o tempo de chegada à origem (seriam servidos juntos depois); se não, é somado também o tempo necessário para efetivamente servir aquele pedido (tempo origem-destino)
    - O tempo a ser somado é calculado de uma forma relativamente constante:

X * LIFT_SPEED + Y * STOP_TIME

(X = Distância a percorrer, Y = Nº pisos destino distintos na tarefa)



Aqui:

- *SmallestTimeEstimate* estima o piso de destino: se o pedido tiver direção ascendente, então o destino estimado será metade da diferença entre o número de pisos e o piso de origem do pedido; caso contrário, é estimado que o destino é o piso 0
- *SmallestTimeNumpad* utiliza diretamente o piso de destino do pedido

Outros mecanismos - Descoberta de agentes

Lift Agent:

- O setup(), regista-se como agente que oferece um serviço do tipo “lift”

```
@Override
protected void setup() {
    //Register as a lift so building can later find out about it at runtime
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("lift");
    sd.setName(getLocalName());
    dfd.addServices(sd);
    try {
        DFService.register( agent: this, dfd);
    } catch (FIPAException fe) {
        fe.printStackTrace();
    }
    //System.out.println("Lift registered!");
    addBehaviour(new CallAnswerer( a: this, MessageTemplate.MatchPerformative(ACLMessage.CFP)));
}
```

Building Agent:

- O setup() realiza uma pesquisa inicial dos agentes que oferecem o serviço “lift” (guardando os seus AIDs num buffer local). Adiciona ainda uma subscrição para ser alertado de elevadores que só surjam mais tarde

```
@Override
protected void setup() {
    initialLiftAgentSearch();
    liftAgentSubscription();
    recaller();
    System.out.println("Setting up building");
}

private DFAgentDescription getDFAgentDescriptionTemplate() {
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("lift");
    template.addServices(sd);
    return template;
}

private void initialLiftAgentSearch() {
    DFAgentDescription template = getDFAgentDescriptionTemplate();
    try {
        DFAgentDescription[] result = DFService.search( agent: this, template);
        System.out.println(result.length);
        for (DFAgentDescription aResult : result)
            System.out.println(this.lifts.add(aResult.getName()));
    } catch (FIPAException fe) {
        fe.printStackTrace();
    }
}

private void liftAgentSubscription() {
    DFAgentDescription template = getDFAgentDescriptionTemplate();
    addBehaviour(new LiftAgentSubscription( agent: this, template));
}
```

Software utilizado c/ detalhes relevantes

- SAJaS - Pela ligação a um simulador, permitindo simultaneamente a programação de agentes via classes JADE
- Repast3
  - Simulação de MAS, apresentando a noção de tempo (*ticks*) essencial para programar ações periódicas, algumas métricas a medir e para a lógica de avaliação de pedidos por parte do elevador
  - Fácil parametrização de variáveis independentes
  - Interface integrada para visualização de agentes e interações e coleção de dados para efeitos estatísticos

```
private void buildSchedule() {
    // Build the schedule with actions to be performed on a regular basis
    getSchedule().scheduleActionAtInterval( v: 1, dsurf, s: "updateDisplay", Schedule.LAST);
    getSchedule().scheduleActionAtInterval(CALL_FREQUENCY, () → { building.newCall(); });
    getSchedule().scheduleActionAtInterval(LIFT_SPEED, () → {
            for (LiftAgent agent : agentList) {
                agent.updatePosition();
                agent.executeTasks();
            }
    });
    getSchedule().scheduleActionAtInterval(LIFT_SPEED, () → { recorder.record(); });
    getSchedule().scheduleActionAtEnd(recorder, s: "writeToFile");
}
```

```
private int LIFT_MAX_CAPACITY = 6;
private int CALL_FREQUENCY = 100;
private int NUM_FLOORS = 5;
private int NUM_LIFTS = 4;
private int LIFT_SPEED = 15;
private int LIFT_STRATEGY = 0;
private int CALL_STRATEGY = 0;
//...
@Override
public String[] getInitParam() {
    return new String[]{"CALL_FREQUENCY", "CALL_STRATEGY", "LIFT_MAX_CAPACITY", "LIFT_SPEED", "LIFT_STRATEGY", "NUM_FLOORS", "NUM_LIFTS"};
}
```

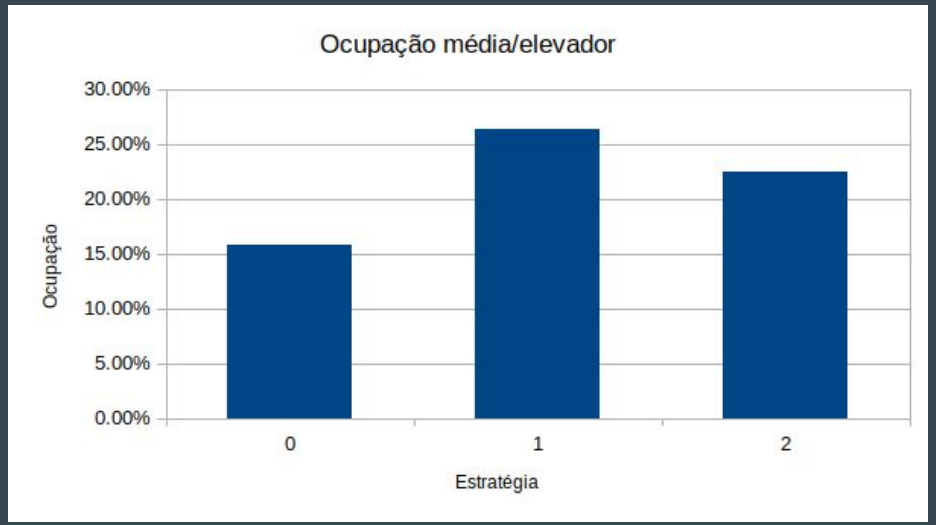
```
private void buildModel() {
    //create space, launch agents

    recorder = new DataRecorder( s: "./data.txt",  simModel: this);
    for (LiftAgent a : agentList) {
        recorder.addNumericDataSource( s: a.getLocalName() + "_ocupation", () -> {
                    if (a.getCurrentTask() != null && !a.isGoingToOrigin())
                        return a.getCurrentTask().getNumAllPeople() * 1.0 / LIFT_MAX_CAPACITY;
                    return 0;
                },
                p: -1, //Record all digits pre decimal separator
                n: 3); //Round to 3 digits post decimal separator
        recorder.addNumericDataSource( s: a.getLocalName() + "_usage_rate", a::getUsageRate,  p: -1,  n: 3);
        recorder.addNumericDataSource( s: a.getLocalName() + "_min_call_time", a::getMinWaitingTime,  p: -1,  n: 3);
        recorder.addNumericDataSource( s: a.getLocalName() + "_max_call_time", a::getMaxWaitingTime,  p: -1,  n: 3);
    }

    recorder.addNumericDataSource( s: "global_min_call_time",
            () -> Collections.min(agentList.stream().map(LiftAgent::getMinWaitingTime).collect(Collectors.toList())),  p: -1,  n: 3);
    recorder.addNumericDataSource( s: "global_max_call_time",
            () -> Collections.max(agentList.stream().map(LiftAgent::getMaxWaitingTime).collect(Collectors.toList())),  p: -1,  n: 3);
}
```

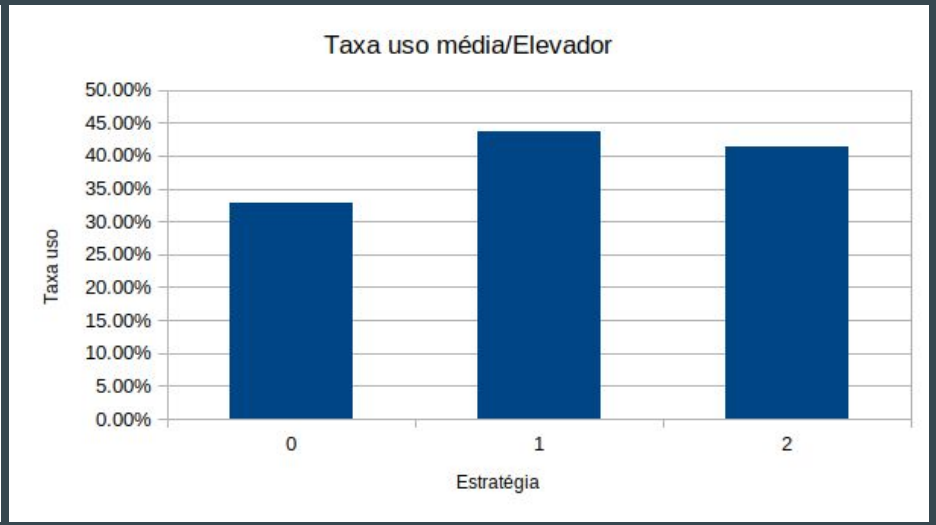
Experiências realizadas (I)

Algumas estatísticas sobre algumas execuções teste. Em todas elas, estes parâmetros foram constantes, variando apenas LIFT_STRATEGY entre os valores 0 (*Closest*), 1 (*SmallestTimeEstimate*) e 2 (*SmallestTimeNumpad*):

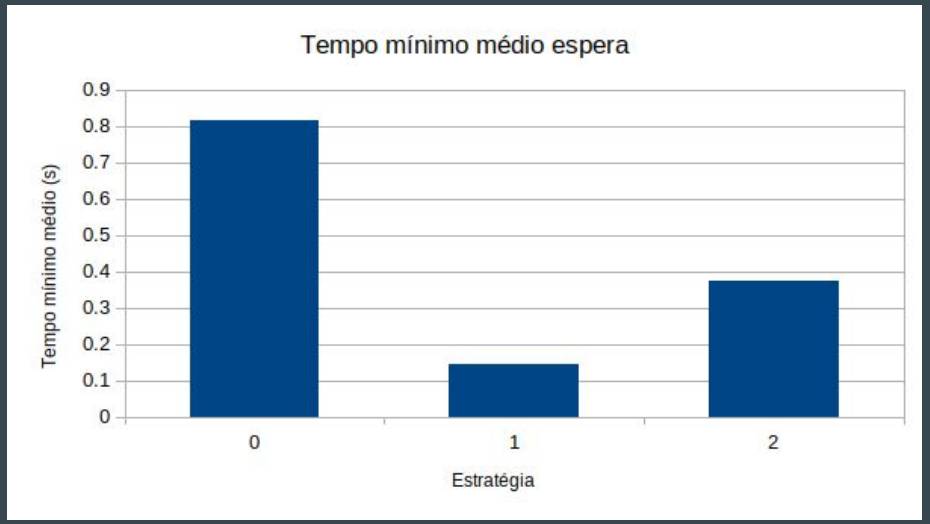
CALL_FREQUENCY: 100 CALL_STRATEGY: 0 (*Morning*) LIFT_MAX_CAPACITY: 6 LIFT_SPEED: 15 NUM_FLOORS: 100 NUM_LIFTS: 10

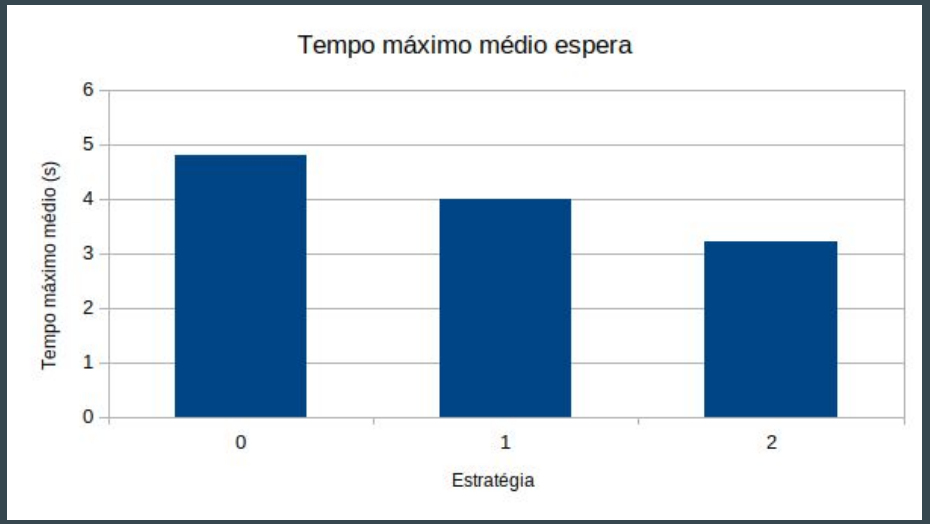
Mediu-se a ocupação média de cada elevador, a taxa de uso média e tempos mínimos e máximos médios de espera por pedido (tempo espera = tempo desde realização do pedido até à colheita do elevador). Os resultados obtidos encontram-se nos gráficos seguintes.





Experiências realizadas (II)





Análise de resultados e conclusões

Resultados experiências

- Não foram feitas experiências extensivas de forma a avaliar detalhadamente as imensas variações do sistema, visto que também não era o foco desta 1ª parte
- Ainda assim, resultados preliminares como os do slide anterior são promissores: as estratégias não *naïve* (i.e. *SmallestTime*) apresentam uma melhoria significativa relativamente à estratégia mais tradicional: o tempo de espera é, em média, mais baixo. Para além disso, considerando um nº de pessoas médio equivalente em cada pedido, existe um melhor aproveitamento da capacidade do elevador.

Conclusão/Trabalho futuro

- O resultado final é satisfatório: o sistema está desenvolvido de uma forma bastante flexível, permitiu uma boa aprendizagem desta nova forma de modelação de sistemas e apresenta resultados otimistas de acordo com o objetivo traçado inicialmente (minimizar o tempo de espera).
- Ainda assim, há sempre melhorias possíveis. Outras estratégias de avaliação poderiam ter sido abordadas. Da mesma forma, outras abordagens para agregação de pedidos (para além da implementada de juntar pedidos com a mesma origem e destinos em sentido semelhante) seriam de interesse em explorar.

Exemplos de execução

Execução omissão:

CALL_FREQUENCY: 100 CALL_STRATEGY: 1 LIFT_MAX_CAPACITY: 6 LIFT_SPEED: 15 LIFT_STRATEGY: 0 NUM_FLOORS: 5 NUM_LIFTS: 4

Execuções realizada para recolha de estatísticas:

CALL_FREQUENCY: 100 CALL_STRATEGY: 0 LIFT_MAX_CAPACITY: 6 LIFT_SPEED: 15 LIFT_STRATEGY: 0, 1 e 2 NUM_FLOORS: 100 NUM_LIFTS: 10

Execuções realizada para testar comportamento com poucos elevadores:

CALL_FREQUENCY: 100 CALL_STRATEGY: 0 e 1 LIFT_MAX_CAPACITY: 6 LIFT_SPEED: 15 LIFT_STRATEGY: 1 NUM_FLOORS: 100 NUM_LIFTS: 10

Execuções realizada para observar comportamento em períodos mais ocupados:

CALL_FREQUENCY: 50 CALL_STRATEGY: 1 e 2 LIFT_MAX_CAPACITY: 5 LIFT_SPEED: 15 NUM_FLOORS: 25 NUM_LIFTS: 4

Outras execuções realizadas:

CALL_FREQUENCY: 75 CALL_STRATEGY: 1 LIFT_MAX_CAPACITY: 8 LIFT_SPEED: 10 LIFT_STRATEGY: 1 e 2 NUM_FLOORS: 200 NUM_LIFTS: 20

Classes implementadas

Principais classes implementadas na realização do trabalho:

- **Repast3Launcher** - Para o lançamento da aplicação com o simulador Repast3 (Launcher.java)
- **ContractNetInitiator/Responder** - Para uso do protocolo CFP na atribuição de pedidos (tarefas) a elevadores (BuildingAgent.java e LiftAgent.java)
- **AchieveREInitiator/Responder** - Para os elevadores requererem ao edifício a reatribuição de um determinado pedido ao edifício por incapacidade suficiente (BuildingAgent.java e LiftAgent.java)
- **SubscriptionInitiator** - Para o edifício ser alertado de elevadores que apenas entrem em operação (BuildingAgent.java)