INSTICC - Technical Program Constuctor

Desenvolvido por Bernardo Mota (201900947)

Projeto Orientado por Prof. Joaquim Filipe

Descrição da Atividade

Desenvolvimento de um programa de inteligência artifical capaz de construir horários para as convenções organizadas pelo INSTICC. Este programa aplica conhecimentos obtidos na UC de Inteligência Artificial e, mais especificamente, algoritmos de pesquisa de espaço de estados.

A solução desenvolvida é um módulo de Node.js desenvolvido em JavaScript. Foi desenvolvido entre Maio e Agosto de 2022 com acompanhamento regular e entrega final, completando 162 horas de trabalho.

A minha classificação proposta é de 5/5.

URL: https://github.com/neurodraft/INSTICC_TPC/

Relatórios:

Técnicas Aplicadas - 28/06/2022

Este documento representa uma exposição, explicação e reflexão das técnicas aplicadas, até ao momento, no desenvolvimento de uma inteligência artifical capaz de construir horários adequados para a conferência ICEIS, respeitando restrições fortes e maximizando preferências suaves estabelecidas.

Implementação genéricas de A*

O algoritmo A*, na sua variação original, foi implementado genéricamente utilizando uma Priority Queue do tipo Min Heap para lista de nós abertos (ordenação por valor f, garante inserção na posição correta em O(logn) e obtenção em O(1)). [https://github.com/ignlg/heap-js]

É também utilizada uma biblioteca de hashing de objetos [https://www.npmjs.com/package/object-hash] para gerar hashes correspondentes ao estado e facilmente comparar com outros estados existentes em abertos ou fechados. É mantido em paridade com a Priority Queue de nós abertos um mapa entre hash criptográficas de estados e o nó aberto correspondente para rapidamente verificar e obter um nó previamente criado sem necessidade de percorrer o lista de prioridade. O conjunto de nós fechados é também um mapa entre hash e nó.

É possível configurar um tempo limite para a execução do algoritmo, chaves do objeto do estado que se pretende ignorar na comparação de estados, frequência de reportagem de progresso e uma função de reportagem de progresso.

Esta implementação encontra-se em `algorithms\astar.js`. Foi testada com o desenvolvimento de um simples sokoba na pasta `experimentation`.

Implementação genéricas de Depth-First, variação Sucessores Iterativos

Prevendo a necessidade de produzir incrementalmente os sucessores de um nó em expansão, de forma a permitir a progressão para níveis mais fundos em tempo útil em situações em que o número de possíveis sucessores é regularmente na ordem dos milhares para cada novo nó como é o caso das possíveis colocações de qualquer grupo de artigos em qualquer sessão disponível, foi imaginada uma variação adaptável a maioria dos algoritmos de procura em grafos.

A única diferença na interface do algoritmo é a utilização de uma função de sucessores iterativa, ou seja, em vez de obter todos os sucessores obtem-se o próximo disponível até retornar NULL, e um máximo de sucessores a gerar por iteração. Na variação do Depth-First implementada, enquanto um nó não ficar sem novos sucessores disponíveis ao fim de uma iteração continua a ser reintroduzido na lista de nós abertos imediatamente antes dos seus filhos.

Esta implementação encontra-se em `algorithms\iterative-sucessors-df.js`.

Agrupador Iterativo de Artigos

Foi implementado um agente auxiliar à função iterativa de geração de sucessores, adaptado ao domínio do problema e o algoritmo ainda não se encontra separado do negócio. No entanto pretendo futuramente generalizar o algoritmo desenvolvido e formaliza-lo devidamente.

Este agente ao ser criado começa por gerar uma matrix de distâncias entre os artigos, a tabela resultante é depois usada para look-up. A função de distância aplicada de momento soma as distância de manhattan (L1 form) entre os diferentes vetores binários e soma-os com pesos, da seguinte forma:

> $D(p1, p2) = d_1(p1.areas, p2.areas) + d_1(p1.topics, p2.topics) * 0.5 + d_1(p1.restrictions, p2.restrictions) * 0.25 + d_1(p1.preferences, p2.preferences) * 0.25$

De seguida cria um nó para cada combinação única entre dois artigos, com a distância calculada previamente e uma lista indefinida de nós filhos - guarda-os numa lista de nós iniciais. Conclui assim a fase de "aquecimento".

O agente dispobiliza uma função que devolve o próximo grupo com maior semelhança para uma chave de consumidor, uma função de validação e uma função de objetivo.

No primeiro pedido de cada consumidor é registada a sua chave e criada uma nova lista de nós abertos só para aquele consumidor (usando novamente uma Priority Queue ordenada por menor distância), a lista de nós iniciais gerados no "aquecimento" é filtrada usando a função de validação e os elementos do resultado são inseridos na fila de nós abertos do consumidor.

De seguida o algoritmo inicia a procura pelo primeiro (melhor) nó que cumpra a função objectivo, cortanto os ramos que não cumpram a função de validação. Mais especificamente:

- Remove-se o nó de melhor distância na lista de abertos;
- Verifica-se se o nó já foi expandido previamente (existência de filhos gerados):
    - Caso o nó ainda não esteja expandido, o agente expande **todos** os sucessores possíveis para o nó (independentemente do consumidor). Os grupos são tratados como combinações (e não arranjos), logo ao gerar pela primeira vez um sucessor, é primeiro verificado se esta combinação já existe num mapa de correspondência entre identificadores únicos de combinação e nós:

- Agora que está garantido que o nó está expandido, percorre-se a lista de filhos e adicionam-se os válidos para o consumidor à fila de nós abertos deste;
- Verifica-se se o melhor nó previamente removido de abertos é objetivo:
    - Se for, é devolvido como próximo melhor conjunto para aquele consumidor.
- Da próxima vez que o mesmo consumidor ativar o agente, o algoritmo irá continuar a partir do ponto onde ficou e garantir que devolve sempre o próximo nó de menor distância (imporante gerar sucessores antes de retornar nó objetivo).

Esta implementação utiliza ponteiros/referências para partilhar o mesmo grafo entre múltiplos consumidores, cada um com a sua própria fila de nós abertos.

No domínio do problema, a geração de sucessores consiste em todas as possíveis junções de qualquer um novo artigo à combinação que está a ser expandida. A distância resultante do novo grupo é o máximo valor entre a distância do nó pai e as distância entre o novo artigo e cada um dos restantes do grupo. É também adicionado ao nó a duração total do grupo, vetores combinados de areas, de tópicos, de restrições e de preferências. A função de validade verifica se o sucessor não tem restrição para a sessão geral a ser preenchida, só combina artigos na lista de artigos disponíveis e que não excede a duração a preencher. A função objetivo apenas verifica se o conjunto tem mais do que a duração mínima para a sessão.

O agente encontra-se em `tpc_domain\paper-grouping-agent.js`.

Resultados da aplicação de Depth-First com Sucessores Iterativos ao problema com auxilio do agente agrupador de artigos

Foi testada a geração de um horário para os artigos e sessões da ICEIS2022 usando o algoritmo depth-first com sucessores iterativos, em que a função geradora de sucessores regista um consumidor com o agente para cada possível sessão a preencher no estado e altera rotativamente entre a colocação do próximo melhor grupo para cada sessão. Quando uma sessão esgota possíveis sucessores é removida de rotação e quando não restam sessões com novos sucessores considera-se o nó inteiramente expandido. O script de teste é `tpc_domain\DFISTesting.js`.

Com um máximo de apenas 1 sucessor por expansão, foi obtido o resultado `tpc_domain\results\dfISResult01.txt` que foi validado por uma função de avalição desenvolvida em `tpc_domain\evaluateMatches.js`. A solução devolvida em 8,6 segundos é válida face às restrições (ainda não considerando a restrição forte do mesmo indivíduo não poder estar presente simulaneamente em mais do que uma sala) e garante a preferência de sessão a dois artigos. Apenas gera 29 nós e não chega a expandir totalmente nenhum deles.

Implementação genéricas de A*, variação Sucessores Iterativos

Semelhante às alterações adicionadas ao algoritmo depth-first para suportar sucessores iterativos, foi desenvolvido uma variação também do A*.

A principal diferença ocorre na reinserção do nó em abertos após uma expansão parcial (ainda existem sucessores). Neste caso, os sucessores e o nó pai são reinseridos na Priority Queue de abertos e imediatamente ordenados por custo, no caso de todos os sucessores expandidos serem piores que o pai então este volta a expandir parcialmente pois permanece o nó de menor valor f.

Esta implementação encontra-se em `algorithms\iterative-sucessors-astar.js`.

Resultados da aplicação de A* com Sucessores Iterativos ao problema com auxilio do agente agrupador de artigos

Foi testada a geração de um horário para os artigos e sessões da ICEIS2022 usando o algoritmo A* com sucessores iterativos, com uma função geradora de sucessores identica à descrita para depth-first. O script de teste é `tpc_domain\aStarISTesting.js`.

Neste momento ainda estou a decidir como avançar relativamente ao custo de transição e à função heurística. Comecei por utilizar a distância do grupo inserido como sendo o custo de transição, mas, sem uma heurística muito bem adequada para este custo de distância, o algoritmo ao encontrar o momento em que o custo real supera o custo previsto, parmanece "preso" na expansão de todos os possíveis sucessores nesse limiar durante horas. Admito que ainda não consegui raciocinar claramente sobre esta relação entre o custo de transição e a heurística para o problema em mão.

A heurística desenvolvida em `tpc_domain\tpcDomain.js` consiste de momento na soma de:

- O número de sessões por preencher;
- O produto interno entre o vector binário de restrições nos artigos por colocar e o vector binário de sessões gerais ainda por preencher (incidências);
- O número de artigos com preferências mas sem nenhuma ainda garantida.

Com um custo de transição fixo de 1 (profundidade) e a heurística descrita acima foi possível rápidamente obter um resultado melhor que o obtido no depth-first:

- Com um máximo de 8 sucessors por expansão, foi obtido o resultado `tpc_domain\results\aStarISResult_fixedCost_8mspi`. A solução devolvida em 15,8 segundos é válida face às restrições (ainda não considerando a restrição forte do mesmo indivíduo não poder estar presente simulaneamente em mais do que uma sala) e garante a preferência de sessão a 3 artigos. Gera 219 nós e expande totalmente apenas 1.

- Com um máximo de 128 sucessors por expansão, foi obtido o resultado `tpc_domain\results\aStarISResult_fixedCost_128mspi`. A solução devolvida em 85,1 segundos é válida face às restrições (ainda não considerando a restrição forte do mesmo indivíduo não poder estar presente simulaneamente em mais do que uma sala) e garante a preferência de sessão a 5 artigos. Gera 3465 nós e expande totalmente apenas 1.

Relatório Final 30/08/22

Foram expostas as configurações relevantes para facilmente se alterar o comportamento do TPC, por exemplo:

```
tpcConfig = {
    // Base cost added to g with each transition
    baseTransitionCost: 1,
    // Session evaluation penalty rules
    sessionPenalties: {
        // Base penalty for sessions without common topics
        noCommonTopicsPenalty: 4,
        // Multiplier for common topics total beyond first common topic
        commonTopicsBeyondFirstPenaltyMultiplier: 2,
```

```
        // Multiplier for areas total beyond first area in group...
        areasBeyondFirstPenaltyMultiplier: {
            // ...when there aren't any topics in common
            withoutCommonTopics: 8,
            // ...when there is at least a topic in common
            withCommonTopics: 2,
        },
        // Multiplier for total of duplicate simultaneous areas in
different rooms
        simultaneousSessionsAreaSimilarityPenaltyMultiplier: 1,
        simultaneousSessionsCommonTopicSimilarityPenaltyMultiplier: 8,

        // Multiplier for undertime normalized relative to Session Duration
        // eg.: 15 / 60 = 0.25 * 10 = 2.5 penalty score added for being 15
minutes under in a 60 minutes session
        // eg.: 15 / 105 = ~0.143 * 10 = ~1.43 penalty score added for
being 15 minutes under in a 105 minutes session
        undertimeNormalizedPenaltyMultiplier: 80,

        // Multiplier for product between overtime and Session Duration
(gets larger as any of the two increases).
        // eg.: 5 * 60 = 300 * 0.025 = 7.5 penalty score added for going 5
minutes over in a 60 minutes session
        // eg.: 5 * 105 = 525 * 0.025 = 13.125 penalty score added for
going 5 minutes over in a 105 minutes session
        overtimeSessionDurationProductPenaltyMultiplier: 0.0125
    },
    // Valid paper group durations for given general session duration
    validDurations: [
        {
            sessionDuration: 60,
            validDurations: [45, 50, 55, 60, 65]
        },
        {
            sessionDuration: 75,
            validDurations: [60, 65, 70, 75, 80]
        },
        {
            sessionDuration: 90,
            validDurations: [75, 80, 85, 90, 95]
        },
        {
            sessionDuration: 105,
            validDurations: [90, 95, 100, 105, 110]
        },
        {
            sessionDuration: 120,
            validDurations: [105, 110, 115, 120, 125]
        },

    ],
    // Heuristic cost multipliers
    heuristicMultipliers: {
        // Multiplier for remaining papers total
```

```
        remainingPapersMultiplier: 1,
        // Multiplier for unmet preferences total
        unmetPreferencesMultiplier: 1
    },
    iterativeAStar: {
        // Additional cost for each reexpansion of a node
        nodeReExpansionAdditionalCost: 20,
        // JavaScript expression for determining how many sucessors to
expand at once in function of d (depth)
        maxSuccessorsPerIterationExpression:
            //"Math.max(16 / Math.pow(2, d), 4)"
            //"Math.max(32 / Math.pow(2, d), 4)"
            //"Math.max(64 / Math.pow(2, d), 4)"
            //"Math.max(128 / Math.pow(2, d), 4)"
            //"Math.max(256 / Math.pow(2, d), 4)"
            //"Math.max(512 / Math.pow(2, d), 4)"
            "Math.max(1024 / Math.pow(2, d), 4)"
            //"2048"
            //"Math.max(256 / (d + 1), 8)"
            //"Math.max(64 / (d + 1), 4)"
    }

};
```

Foram removidas algumas operações dispendiosas que ocorriam na geração de todos os sucessores relacionadas com geração de chaves de consumidor através de hashing. Após aplicar as mesmas simplificações ao hashing dos estados no A*, o algoritmo começou a cortar ramos redundantes, melhorando execuções prévias de 15 minutos para a menos de 1 minuto.

O algoritmo consegue agora resolver horários até um maior nível de qualidade em tempo útil, no entanto esta melhoria gerou um maior consumo de memória virtual.

É possível agora efetuar testes utilizando o seguinte script: https://github.com/neurodraft/INSTICC_TPC/blob/master/tpc_domain/tpcTesting.js. Os resultados obtidos estão disponibilizados na seguinte pasta: https://github.com/neurodraft/INSTICC_TPC/tree/master/tpc_domain/results.