UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Influence-based Motion Planning Algorithms for Games

Gonçalo Nuno Paiva Amador

Tese para obtenção do Grau de Doutor em
Engenharia Informática
(3º ciclo de estudos)

Orientador: Prof. Doutor Abel João Padrão Gomes

Covilhã, Junho of 2017

This thesis was conducted under supervision of Professor Abel João Padrão Gomes. The research work behind this doctoral dissertation was developed within the NAP-Cv (Network Architectures and Protocols - Covilhã) at Instituto de Telecomunicações, University of Beira Interior, Portugal.

The thesis was funded by the Portuguese Research Agency, *Foundation for Science and Technology (Fundação para a Ciência e a Tecnologia)* through grant contract SFHR/BD/86533/2012 under the Human Potential Operational Program Type 4.1 Advanced Training POPH (Programa Operacional Potencial Humano), within the National Strategic Reference Framework QREN (Quadro de Referência Estratégico Nacional), co-funded by the European Social Fund and by national funds from the Portuguese Education and Science Minister (Ministério da Educação e Ciência).

À minha família e amigos.

To my family and friends.

Acknowledgments

This thesis would not have reached this final stage without my supervisor, my family, and my friends.

To my supervisor Professor Abel Gomes. First, for his guidance in helping me defining a proper work plan for my PhD thesis. Second, by his interesting doubts and suggestions, that often resulted in extra programming and writing hours. Third, for his rigorous (and I really mean 'rigorous') revision of every single article related to my PhD and thesis chapters/sections. Finally, for accepting, enduring, and surviving being my supervisor, specially because I am sometimes difficult to deal with.

To my closest family, namely, my parents José Amador and Glória Amador, my brother Álvaro Amador, my deceased uncle Neves, my aunt Estrela, and their soon Nuno Neves and his family. To my parents, for making many efforts and sacrifices along the years of my life, that (among many other things) allowed me to conclude my bachelors degree. Also, to the fact that they do not annoy their son, now in his thirties, with the fact of him (me) still being a "student" in academia, and not already "working". To the rest of my family, for their interest and preoccupation with my happiness and life in general.

To my friend Fernando Neves (non relative), for his daily talks related with my thesis and miscellaneous subjects. For sharing his advice and experience during his already concluded PhD thesis. To my friend Tiago Ferreira, for our conversations and exchanges of ideas related to my PhD thesis, to his MSc thesis, and miscellaneous subjects. To my friend Sérgio da Piedade, for the discussions we had regarding our respective PhD theses and miscellaneous subjects.

To my friends, namely, Ana Inês Rodrigues, Pedro Guilherme, Daniela Jesus, and Orlando Pereira for the daily talks about anything beyond my thesis, that allowed to recover energies and to tackle problems from different directions. In particular, to my friends Daniela Jesus, and Orlando Pereira for their availability and the effort they put in reading and appreciating my articles, prior to being delivered for revision by my supervisor, providing my with an insight from people outside this thesis specific knowledge/subject.

Agradecimentos

A presente tese não teria chegado a esta fase final sem o meu supervisor, a minha família, e os meus amigos.

Ao meu orientador Professor Abel Gomes. Primeiro, pela sua orientação e ajuda na definição de um plano de trabalhos adequado à realização da minha tese de doutoramento. Em segundo lugar, pelas suas dúvidas e sugestões interessantes que, muitas vezes, resultaram em várias horas adicionais de programação e escrita. Em terceiro lugar, pelas suas rigorosas revisões (de frisar o 'rigorosas') de cada artigo relacionado com o meu doutoramento, bem como de cada capítulo da tese. Finalmente, por aceitar, aguentar e sobreviver a ser o meu orientador, sendo eu por vezes alguém com quem é difícil lidar.

À minha família mais próxima, ou seja, aos meus pais José Amador e Glória Amador, ao meu irmão Álvaro Amador, ao meu já falecido tio Neves, à minha tia Estrela, e ao seu filho Nuno Neves e respectiva família, por me apoiarem nesta etapa da minha vida. Aos meus pais pelos muitos esforços e sacrifícios ao longo dos anos da minha vida, que (entre muitas outras coisas) permitiram concluir a minha licenciatura, além de não incomodarem o seu filho, agora nos seus trinta, por ainda ser um "estudante" e não estar a "trabalhar". Ao resto da minha família pelo seu interesse e preocupação com a minha felicidade e vida em geral.

Ao meu amigo Fernando Neves, pelas suas conversas diárias relacionadas com a minha tese entre outros assuntos, e pelos conselhos e experiência que me transmitiu no decorrer dos trabalhos dedoutoramento. Ao meu amigo Tiago Ferreira, pelas nossas conversas e trocas de ideias relacionadas com a minha tese de doutoramento, com a sua tese de mestrado, entre outros assuntos. Ao meu amigo Sérgio da Piedade pelas discussões que tivemos sobre as nossas respectivas teses e outros assuntos. Aos meus amigos Ana Inês Rodrigues, Pedro Guilherme, Daniela Jesus e Orlando Pereira, entre outros, pelas conversas diárias sobre outras coisas que não a minha tese, que permitiram recuperar energias e abordar novas formas de resolver problemas. Em particular, aos meus amigos Daniela Jesus e Orlando Pereira pela disponibilidade e esforço na apreciação dos meus artigos, pré-revisão por parte do meu orientador, facultando uma visão de alguém fora do tema da minha tese.

Abstract

In games, motion planning has to do with the motion of non-player characters (NPCs) from one place to another in the game world. In today's video games there are two major approaches for motion planning, namely, *path-finding* and *influence fields*.

Path-finding algorithms deal with the problem of finding a path in a weighted search graph, whose nodes represent locations of a game world, and in which the connections among nodes (edges) have an associated cost/weight. In video games, the most employed pathfinders are A* and its variants, namely, Dijkstra's algorithm and best-first search. As further will be addressed in detail, the former pathfinders cannot simulate or mimic the natural movement of humans, which is usually without discontinuities, i.e., smooth, even when there are sudden changes in direction.

Additionally, there is another problem with the former pathfinders, namely, their lack of adaptivity when changes to the environment occur. Therefore, such pathfinders are not adaptive, i.e., they cannot handle with search graph modifications during path search as a consequence of an event that happened in the game (e.g., when a bridge connecting two graph nodes is destroyed by a missile).

On the other hand, influence fields are a motion planning technique that does not suffer from the two problems above, i.e., they can provide smooth human-like movement and are adaptive. As seen further ahead, we will resort to a differentiable real function to represent the influence field associated with a game map as a summation of functions equally differentiable, each associated to a repeller or an attractor. The differentiability ensures that there are no abrupt changes in the influence field, consequently, the movement of any NPC will be smooth, regardless if the NPC walks in the game world in the growing sense of the function or not. Thus, it is enough to have a spline curve that interpolates the path nodes to mimic the smooth human-like movement.

Moreover, given the nature of the differentiable real functions that represent an influence field, the removal or addition of a repeller/attractor (as the result of the destruction or the construction of a bridge) does not alter the differentiability of the global function associated with the map of a game. That is to say that, an influence field is adaptive, in that it adapts to changes in the virtual world during the gameplay.

In spite of being able to solve the two problems of pathfinders, an influence field may still have local extrema, which, if reached, will prevent an NPC from fleeing from that location. The local extremum problem never occurs in pathfinders because the goal node is the sole global minimum of the cost function. Therefore, by conjugating the cost function with the influence function, the NPC will never be detained at any local extremum of the influence function, because the minimization of the cost function ensures that it will always walk in the direction of the goal node. That is, the conjugation between pathfinders and influence fields results in movement planning algorithms

which, simultaneously, solve the problems of pathfinders and influence fields.

As will be demonstrated throughout this thesis, it is possible to combine influence fields and A*, Dijkstra's, and best-first search algorithms, so that we get hybrid algorithms that are adaptive. Besides, these algorithms can generate smooth paths that resemble the ones traveled by human beings, though path smoothness is not the main focus of this thesis. Nevertheless, it is not always possible to perform this conjugation between influence fields and pathfinders; an example of such a pathfinder is the fringe search algorithm, as well as the new pathfinder which is proposed in this thesis, designated as best neighbor first search.

Keywords

Path-finding, Influence Maps, Motion Planning, Path Planning, Navigation Problem, Spatial Reasoning.

Resumo

Em jogos de vídeo, o planeamento de movimento tem que ver com o movimento de NPCs ("Non-Player Characters", do inglês) de um lugar para outro do mundo virtual de um jogo. Existem duas abordagens principais para o planeamento de movimento, nomeadamente *descoberta de caminhos* e *campos de influência*.

Os algoritmos de descoberta de caminhos lidam com o problema de encontrar um caminho num grafo de pesquisa pesado, cujos nós representam localizações de um mapa de um jogo, e cujas ligações (arestas) entre nós têm um custo/peso associado. Os algoritmos de descoberta de caminhos mais utilizados em jogos são o A* e as suas variantes, nomeadamente, o algoritmo de Dijkstra e o algoritmo de pesquisa do melhor primeiro ("best-first search", do inglês). Como se verá mais adiante, os algoritmos de descoberta de caminhos referidos não permitem simular ou imitar o movimento natural dos seres humanos, que geralmente não possui descontinuidades, i.e., o movimento é suave mesmo quando há mudanças repentinas de direcção.

A juntar a este problema, existe um outro que afeta os algoritmos de descoberta de caminhos acima referidos, que tem que ver com a falta de adaptatividade destes algoritmos face a alterações ao mapa de um jogo. Ou seja, estes algoritmos não são adaptativos, pelo que não permitem lidar com alterações ao grafo durante a pesquisa de um caminho em resultado de algum evento ocorrido no jogo (e.g., uma ponte que ligava dois nós de um grafo foi destruída por um míssil).

Por outro lado, os campos de influência são uma técnica de planeamento de movimento que não padece dos dois problemas acima referidos, i.e., os campos possibilitam um movimento suave semelhante ao realizado pelo ser humano e são adaptativos. Como se verá mais adiante, iremos recorrer a uma função real diferenciável para representar o campo de influência associado a um mapa de um jogo como um somatório de funções igualmente diferenciáveis, em que cada função está associada a um repulsor ou a um atractor. A diferenciabilidade garante que não existem alterações abruptas ao campo de influência; consequentemente, o movimento de qualquer NPC será suave, independentemente de o NPC caminhar no mapa de um jogo no sentido crescente ou no sentido decrescente da função. Assim, basta ter uma curva spline que interpola os nós do caminho de forma a simular o movimento suave de um ser humano.

Além disso, dada a natureza das funções reais diferenciáveis que representam um campo de influência, a remoção ou adição de um repulsor/atractor (como resultado da destruição ou construção de uma ponte) não altera a diferenciabilidade da função global associada ao mapa de um jogo. Ou seja, um campo de influência é adaptativo, na medida em que se adapta a alterações que ocorram num mundo virtual durante uma sessão de jogo.

Apesar de ser capaz de resolver os dois problemas dos algoritmos de descoberta de

caminhos, um campo de influência ainda pode ter extremos locais, que, se alcançados, impedirão um NPC de fugir desse local. O problema do extremo local nunca ocorre nos algoritmos de descoberta de caminhos porque o nó de destino é o único mínimo global da função de custo. Portanto, ao conjugar a função de custo com a função de influência, o NPC nunca será retido num qualquer extremo local da função de influência, porque a minimização da função de custo garante que ele caminhe sempre na direção do nó de destino. Ou seja, a conjugação entre algoritmos de descoberta de caminhos e campos de influência tem como resultado algoritmos de planeamento de movimento que resolvem em simultâneo os problemas dos algoritmos de descoberta de caminhos e de campos de influência.

Como será demonstrado ao longo desta tese, é possível combinar campos de influência e o algoritmo A*, o algoritmo de Dijkstra, e o algoritmo da pesquisa pelo melhor primeiro, de modo a obter algoritmos híbridos que são adaptativos. Além disso, esses algoritmos podem gerar caminhos suaves que se assemelham aos que são efetuados por seres humanos, embora a suavidade de caminhos não seja o foco principal desta tese. No entanto, nem sempre é possível realizar essa conjugação entre os campos de influência e os algoritmos de descoberta de caminhos; um exemplo é o algoritmo de pesquisa na franja ("fringe search", do inglês), bem como o novo algoritmo de pesquisa proposto nesta tese, que se designa por algoritmo de pesquisa pelo melhor vizinho primeiro ("best neighbor first search", do inglês).

Palavras-chave

Pesquisa de Caminhos, Campos de Influência, Planeamento de Movimento, Planeamento de Rotas, Problema de Navegação, Planeamento Espacial.

Resumo Alargado

O trabalho realizado no âmbito desta tese tem a ver com o desenvolvimento de técnicas e algoritmos de planeamento de movimento no contexto de jogos de vídeo. Especificamente, pretende-se mostrar que é viável integrar campos de influência com algoritmos de descoberta de caminhos e, simultaneamente, eliminar o problema do extremo local associado aos mapas de influência. Os algoritmos de descoberta de caminhos A* [HNR68], Dijkstra [Dij59], e ainda pesquisa pelo melhor primeiro ("best-first search", do inglês) [Poh70b] serão os principais algoritmos utilizados na integração com mapas de influência. Será também demonstrado que nem todos os algoritmos de descoberta de caminhos podem ser reformulados de forma a serem integrados com mapas de influência; por exemplo, um desses algoritmos é descrito nesta tese, e é designado por algoritmo de pesquisa aparada pelo melhor vizinho primeiro ("trimmed best-neighbor first search", do inglês, ou apenas TBNFS), que é um algoritmo de pesquisa óptimo. Além disso, será demonstrado que é possível elaborar algoritmos de descoberta de caminhos exclusivamente com custos baseados em influência, em vez de custos baseados em distância.

Planeamento de Movimento em Jogos de Vídeo

A pesquisa em grafos é um tópico importante em inteligência artificial [PM10]. A pesquisa em grafos é conhecida como *planeamento de movimento* em robótica e vídeo jogos [LaV06], e tem que ver com o encontrar de um caminho num grafo que descreve as posições de nós passáveis de um mapa de um jogo. Recorde-se que a inteligência artificial é um ramo de conhecimento de ciências da computação que foi definido por John McCarty em 1956 como "a ciência e engenharia de máquinas inteligentes" [McC07], existindo definições alternativas, mas semelhantes, na literatura [RN03].

Em jogos, o planeamento de movimento é utilizado para dotar NPCs com um comportamento percebido como inteligente [BS04]; especificamente, o planeamento de movimento tem por objetivo encontrar caminhos para que NPCs se movam de um lugar para outro num mapa de um jogo. Doravante, o planeamento de caminhos será sempre discutido no contexto dos jogos de vídeo, pelo que importa desde já referir que o planeamento de movimento se divide em duas técnicas principais: *descoberta de caminhos* e *campos de influência* [Rab02, Paa08].

Algoritmos de Descoberta de Caminhos

A descoberta de caminhos desempenha um papel importante em muitas áreas de conhecimento, incluindo jogos de vídeo [YS08] [VX09], roteamento em redes de comunicação

[TS01] [TPA12], planeamento de movimento em robótica [Ark86] [LS90] e navegação através do sistema de posicionamento global (do inglês "Global Positioning System" ou apenas GPS) [SSP08]. Um algoritmo de descoberta de caminhos refere-se a um algoritmo que encontra uma rota (se existir) entre um nó de início e um nó de destino num grafo de pesquisa pesado, o que significa que um algoritmo de descoberta de caminhos é completo.

Além disso, a maioria dos algoritmos de descoberta de caminhos asseguram que um caminho encontrado é o de menor custo, i.e., óptimo. Podem existir múltiplos caminhos óptimos, o que significa que diferentes algoritmos de descoberta de caminhos óptimos podem encontrar caminhos distintos, mas ainda assim serem caminhos óptimos. O grafo de pesquisa é uma estrutura matemática discreta composta por nós ligados por arestas. Em jogos, estes nós estão relacionados com as coordenadas Cartesianas de um mapa de um jogo. Se uma aresta tem um valor associado (i.e., um custo ou peso), o grafo de pesquisa diz-se pesado. Se as arestas só podem ser percorridas numa única direção, diz-se que o grafo é dirigido. No entanto, esta tese lida exclusivamente com grafos ponderados positivos não-dirigidos.

Em jogos o algoritmo de descoberta de caminhos mais utilizado é o A* ou uma das suas variantes, a saber, o algoritmo de Dijkstra, a pesquisa pelo melhor primeiro ("best-first search", do inglês), e a pesquisa na franja ("fringe search", do inglês). O A*, o Dijkstra e a pesquisa na franja são algoritmos completos e óptimos, enquanto que o pesquisa pelo melhor primeiro é apenas completo. Todos os algoritmos de descoberta de caminhos referidos sofrem de várias limitações:

1. ***Adaptatividade na Descoberta de Caminhos***. Nenhum é capaz de lidar com alterações ao grafo durante a pesquisa de um caminho, i.e., não são *algoritmos de descoberta de caminhos adaptativos*. De lembrar que as mudanças ao mundo de um jogo incluem: *remoção de nós* (e.g., uma ponte no mundo de um jogo é removida como consequência da sua destruição); *adição de nós ou aresta(s)* (e.g., quando é adicionado um cruzamento que liga duas estradas). De facto, existem muito poucos algoritmos para descoberta de caminhos adaptativos, a saber, o A* com planeamento a longo prazo (do inglês "lifelong planing A*" ou apenas LPA*) [KLF04] e o D* (também conhecido como A* dinâmico) [Ste94, Ste95, KL05]. No entanto, quando ocorrem alterações ao grafo, os algoritmos de descoberta de caminhos adaptativos requerem mais armazenamento em memória, mais operações de E/S, e o recálculo adicional do custo de nós que diminuem significativamente o seu desempenho temporal. Consequentemente, os algoritmos de descoberta de caminhos adaptativos não são usados em jogos. Ao invés, quando as referidas alterações ocorrem, um caminho é recalculado entre o nó de partida e o nó de destino mais uma vez [SKY08].

2. ***Desempenho Temporal***. Trabalham em modo "off-line" [Poh73, Kor85, GMN94, Ree99, BEHS05, DSC08, TR10, HBUK12, BB12], i.e., não estão incluídos no ciclo

de eventos de um jogo, pois os seus tempos de processamento diminuem notoriamente a taxa de refrescamento de imagem ("frame rate", do inglês).

3. ***Suavização dos Caminhos***. Sofrem de um problema de *suavização de caminhos* porque produzem caminhos linearmente segmentados que resultam da discretização do mapa do jogo, não sendo por isso esteticamente apelativos [Mar02, Ger06, ASK15]. Este problema pode ser solucionado substituindo o caminho encontrado por um caminho curvo [BMS04], o qual pode ser gerado com recurso a *splines* de aproximação (e.g., *splines* de Bézier) No entanto, a menos que se recorra a splines de interpolação (e.g., a *spline* de interpolação cúbica), esta solução é particularmente dispendiosa em termos computacionais porque introduz mais cálculo computacional de forma a evitar colisões entre o NPC que se move ao longo do caminho curvo e os obstáculos na cena [Rab00].

No trabalho de investigação realizado no decorrer desta tese, abordámos principalmente os dois primeiros problemas referidos acima, a adaptatividade na descoberta de caminhos e o desempenho temporal. Este trabalho foi realizado usando campos de influência. Como observado ao longo desta tese, o problema da suavização de caminhos também pode ser resolvido usando mapas de influência porque os mapas de influência podem ser definidos por funções suaves (ou diferenciáveis), com a vantagem de permitir que os NPCs evitem obstáculos.

Campos de Influência

Ao utilizar um campo de influência no planeamento de movimento, os NPCs são levados a evitar colisões com obstáculos, o que imita o movimento esperado de um ser humano, ao longo de vários ciclos de atualização do estado do jogo. Portanto, não se pode dizer que exista um planeamento de caminho entre um nó inicial e um nó final, mas tão só um processo de convergência em direção ao destino. Um campo associa a cada ponto no espaço e tempo de um jogo uma quantidade: um vetor para *campos potenciais* ou *campos de fluxo* e um valor escalar para *campos* ou *mapas de influência*.

Os campos potenciais imitam o comportamento de um objeto que se move dentro de um campo magnético (definido matemáticamente por um campo vetorial), que resulta da soma de campos potenciais subsidiários. Cada campo subsidiário fluindo de/para um ponto é designado de propagador. Um propagador gera um campo potencial que gradualmente converge para zero [HJ08b]. Dependendo da sua carga, um propagador pode ser atractivo (i.e. objetos movem-se na sua direção) ou repulsivo (i.e. objetos afastam-se dele). Nos jogos, os propagadores podem estar associados a entidades estáticas (e.g. a porta de um edifício) e dinâmicas (e.g., um robô ou "robot" ou apenas "bot", do inglês) que se move pelo mapa de um jogo.

Um mapa de influência é matematicamente definido por um campo escalar. As suas aplicações vão além do uso dos campos de potencial no planeamento de movimento,

porque um mapa de influência também pode ser usado para o raciocínio espacial e temporal, particularmente na análise de terreno, bem como na compreensão de eventos que ocorrem ao longo do tempo [SJ12, OSU+13, LCCFL13].

Em geral, os campos de influência têm várias vantagens em relação aos algoritmos de descoberta de caminhos, a saber:

1. Funcionam em tempo real para um único agente.
2. Funcionam em tempo real para múltiplos agentes. Lembre-se que a descoberta de caminhos só funciona para um único agente e não é uma técnica de planeamento de movimento em tempo real.
3. São adaptativos porque podem lidar com alterações no mapa de um jogo.

Não obstante a sua versatilidade, os *campos* sofrem de duas grandes limitações relativamente aos algoritmos de descoberta de caminhos:

1. Não são óptimos, o que é um mecanismo fundamental na definição do grau de dificuldade em jogos; por exemplo, num jogo de corridas, os algoritmos de descoberta de caminhos podem ser configurados para calcular várias rotas, mais ou menos óptimas, permitindo a um jogador alguma margem para ganhar o jogo.
2. Sofrem de um problema de não terminação (i.e., não são completos) em consequência do problema do extremo local; este problema assume a forma de um problema do mínimo local quando um agente de um jogo se move na direcção de valores decrescentes do campo; em alternativa, pode tomar a forma de um problema de máximo local quando um agente de um jogo se move na direcção de valores crescentes do campo. Quando um agente fica preso num extremo local de um campo de influência, fica ali aprisionado [Goo00], pois na sua vizinhança não há nenhum ponto com valor superior (quando se converge no sentido de valores crescentes) ou nenhum ponto com valor inferior (quando se converge no sentido de valores decrescentes) para onde se possa mover. Nos campos potenciais, este fenómeno ocorre em pontos onde a soma total de forças vetoriais atinge o valor zero.

A Declaração da Tese

Por forma a engendrar algoritmos de descoberta de caminhos adaptativos sem alterar o grafo de pesquisa, seguimos a ideia principal de que é possível integrar campos de influência com algoritmos de descoberta de caminhos e, consequentemente, os seus problemas em simultâneo.

Assim sendo, pode enunciar-se a tese deste trabalho da seguinte maneira:

É possível desenvolver novos algoritmos de planeamento de movimento capazes de conjugar os algoritmos de descoberta de caminhos com campos de influência por forma a resolver o problema do extremo local, o problema do consumo excessivo de memória e tempo em relação aos algoritmos de descoberta de caminhos canónicos, o problema de adaptatividade face às alterações ao grafo de pesquisa, bem como o problema de simulação do movimento suave dos seres humanos.

Como visto algures nesta dissertação, no limite, é mesmo possível projetar um algoritmo de descoberta de caminhos baseado em custos baseados na influência em vez de custos baseados na distância.

Metodologia de Investigação

De forma a demonstrar a validade da declaração de tese, adotámos a seguinte metodologia:

Conjunto de mapas de jogos para teste. Nós decidimos usar o conjunto HOG2 de mapas de jogos disponíveis em `http://www.movingai.com/benchmarks/` [Stu12]. Em particular, usámos 10 mapas do Warcraft 3 (cenários exteriores) e 10 mapas do Dragon Age: Origins (cenários interiores). Estes mapas são estruturados em grelhas 2D de células quadradas. Cada mapa é então codificado como um grafo de pesquisa, cujos nós correspondem a células passáveis e cujas arestas indicam relações de adjacência entre células vizinhas. Note-se que as células brancas (chão) ou verde-azul (pântano) correspondem a células passáveis, enquanto que as células restantes correspondem a obstáculos; essas células não-passáveis não são nós do grafo de pesquisa. O conjunto HOG2 é, assim, a base do nosso trabalho.

Fusão de algoritmos de descoberta de caminhos com campos de influência. Este foi um dos grandes desafios deste trabalho doutoral, pois tivémos que encontrar as respostas para as seguintes perguntas: Será que qualquer algoritmo de descoberta de caminhos é fundível com campos de influência gerados por atratores e repulsores? O que torna possível fundir um algoritmo de descoberta de caminhos com um campo de influência? Como é que essa fusão se traduz numa nova função de custo? Essa fusão funciona apenas bem em algoritmos de descoberta de caminhos que escolhem o próximo nó como o nó de menor custo na fila aberta. Assim, a nossa técnica de fusão funciona bem apenas em algoritmos como o A*, o Dijkstra e a pesquisa pelo melhor primeiro. Mas falha noutros algoritmos de descoberta de caminhos como pesquisa na franja, porque este algoritmo realiza uma pesquisa em profundidade ao invés de o fazer em largura. De referir que um algoritmo de descoberta de caminhos que pode ser combinado com um campo de influência requer a reformulação da função de custo de modo a combinar valores de distância e de influência.

Resolução do problema do extremo local dos campos de influência. Como se sabe, os

campos de influência padecem do problema do extremo local. No início deste trabalho de investigação, acreditávamos que a fusão dos campos de influência com os algoritmos de descoberta de caminhos resolveria esse problema. Na verdade, qualquer algoritmo de descoberta de caminhos força o movimento em direção ao nó de destino. Consequentemente, mesmo que o extremo local de um campo de influência seja encontrado, um determinado NPC não fica preso nesse extremo local, ou seja, continua movendo-se pela ação do algoritmo de descoberta de caminhos.

Redução do consumo de memória e de tempo relativamente a algoritmos canónicos de descoberta de caminhos. Também acreditávamos que o uso de atratores como locais por onde se deve passar em direcção ao destino e os repulsores como locais por onde não se deve passar, ou seja, obstáculos e regiões indesejáveis, reduziria o espaço de pesquisa de algoritmos de descoberta de caminhos sensíveis a campos de influência em relação aos algoritmos canónicos (e.g., o Dijkstra, o A*, a pesquisa pelo melhor primeiro). Adicionalmente, além dos referidos algoritmos canónicos, os testes referentes ao consumo de memória e ao desempenho temporal também incluíram os seguintes algoritmos de descoberta de caminhos: o A* estático ponderado ("static weighted A*", do inglês) [Poh70a], o A* dinâmico ponderado ("dynamic weighted A*", do inglês) [Poh73], pesquisa otimista [TR08] e busca cética [TR10]. A colocação de atratores e repulsores foi feita manualmente por simplicidade, mas é possível fazê-lo de forma automatizada usando, por exemplo, a árvore de conectividade mínima do grafo de um jogo.

Adaptividade na procura de caminhos. A adaptividade na procura de caminhos é controlada pela colocação, remoção, e movimento de atratores e repulsores no mapa de um jogo. Por exemplo, quando uma rua é destruída, os nós passáveis a ela associados passam a ser nós intransponíveis, de maneira que se esses nós fossem parte de um caminho entre dois locais, o caminho deve ser reconstruído localmente. Isso pode ser feito colocando repulsores em ambas as extremidades da rua para evitar a entrada na referida rua, evitando assim a re-computação de todo o caminho desde o nó de início ao nó de destino.

Suavização de caminhos. A discretização do mapa de um jogo através de uma grelha de células faz com que qualquer caminho feito em ziguezague, mesmo quando se usa diagonais (ou seja, usando uma vizinhança de 8 para determinar o próximo nó de um caminho). Para dotar um NPC de movimento que simule o movimento humano, devemos suavizar o caminho, tornando-o um caminho curvo [BMS04]. A solução típica para este problema é usar uma spline aproximada (por exemplo, uma spline de Bézier), mas essa solução geométrica não garante que o caminho não colida com obstáculos na cena [Rab00]. Além disso, esta solução é computacionalmente dispendiosa. A nossa solução consiste em usar uma spline de interpolação cúbica [dB78], que não padece do fenómeno de Runge [Run01]. Além disso, no caso de haver o risco de uma colisão eminente com um obstáculo, podemos colocar um repulsor no referido obstáculo para desviar ligeiramente o caminho de forma automática. Mas, se houver a necessidade de elim-

inar ainda mais as pequenas oscilações da spline de interpolação cúbica quando esta vira à direita ou à esquerda, a nossa solução é recorrer ao polinómio de interpolação cúbica de Hermite [FC80].

Plano de Investigação

O trabalho realizado que culminou nesta dissertação seguiu o seguinte plano:

- Um levantamento exaustivo da literatura relacionada com algoritmos de descoberta de caminhos e campos de influência em jogos de vídeo. O referido levantamento resultou na escrita do Capítulo 2.
- A implementaçãao de uma estrutura de software extensível e modular para algoritmos de descoberta de caminhos. Entre as suas características contam-se as seguintes: a geração automática de labirintos através dos algoritmos de Kruskal ou de Prim; a importação de mapas a partir do repositório de mapas no formato HOG2 disponível em `http://movingai.com/benchmarks/`; a inspeção visual de cada algoritmo; a saída de dados estatísticos (e.g. tempo e ocupação da memória) referentes a um conjunto de algoritmos e a um conjunto de mapas e/ou labirintos. Esta estrutura de software foi utilizada para gerar os resultados apresentados nos Capítulos 3, 4 e 5.
- A implementação e testes da nossa primeira técnica que integra campos de influência com algoritmos de descoberta de caminhos, detalhada no Capítulo 3. Inicialmente, pensou-se fazer esta integração através de uma função de custo que simplesmente somava os custos tradicionais baseados na distância aos valores de influência. A técnica que daí resultou, ainda que publicada em conferência [AAG15], não era satisfatória, porque apenas funcionava adequadamente com o algoritmo de Dijkstra. Finalmente, uma versão melhorada desta técnica foi concebida e implementada de forma a assegurar que o NPC em movimento era capaz de fugir de repulsores e passar por atractores rumo ao local de destino.
- A implementação e testes da nossa segunda técnica detalhada no Capítulo 4. À semelhança da primeira técnica (Capítulo 3), fomos capazes de re-desenhar e re-implementar o A*, o Dijkstra, e a pesquisa pelo melhor primeiro com mapas de influências, mas desta vez sem utilizar custos baseados em distância. No entanto, não foi possível fazer o mesmo para a pesquisa na franja. De facto, alterar o custo dos nós de forma a manipular a prioridade de avaliação do próximo nó apenas funciona em algoritmos baseados na pesquisa em largura como é o caso do A*, do Dijkstra, e da pesquisa pelo melhor primeiro, mas não funciona no algoritmo de pesquisa na franja.
- Em boa verdade, as dificuldades em re-desenhar e re-implementar a pesquisa na franja com custos baseados exclusivamente em influência levou-nos ao algo-

ritmo descrito no Capítulo 5, designado por pesquisa pelo melhor vizinho primeiro ("best neighbor first search", do inglês, ou apenas BNFS). Durante a implementação do BNFS, percebemos que a sua optimilidade poderia ser melhorada, o que deu origem a um novo algoritmo (TBNFS) cujo desempenho ultrapassa o do A*, do Dijkstra, e da pesquisa pelo melhor primeiro, bem como da pesquisa na franja.

- A escrita da presente tese de doutoramento.

Contribuições

O primeiro grande contributo desta dissertação reside na integração de algoritmos canónicos de descoberta de caminhos (i.e. o A*, o Dijkstra e a pesquisa pelo melhor primeiro) com mapas de influência que, simultaneamente, solucionam o problema de adaptividade dos algoritmos canónicos de pesquisa de caminhos, e, em simultâneo, os problemas do extremo local e da não-completude dos campos de influência. De facto, os algoritmos de descoberta de caminhos sensíveis a influência descritos nos Capítulos 3 e 4 apontam para a constituição de uma nova família de algoritmos adaptativos que podem ser usados em jogos, com um ou mais agentes.

O segundo grande contributo desta tese tem a ver com a introdução de dois novos algoritmos de descoberta de caminhos sub-ótimos, a saber, a pesquisa do melhor vizinho primeiro ("best-neighbor first search", do inglês, ou apenas BNFS) e a pesquisa aparada do melhor vizinho primeiro ("trimmed best-neighbor first search", do inglês, ou apenas TBNFS), que muitas vezes têm um menor consumo de memória e melhor desempenho temporal comparativamente aos algoritmos A*. Especificamente, o TBNFS está mais próximo do ótimo do que o algoritmo de pesquisa pelo melhor primeiro utilizado em jogos.

Publicações

O trabalho desta tese resultou em vários artigos, nomeadamente:

- G. Amador, A. Gomes. Influence-Based A* Algorithms. *IEEE Transactions on Computational Intelligence and AI in Games* (under revision).
- G. Amador, A. Gomes. Influence-Centric Function A* Algorithms. *IEEE Transactions on Computational Intelligence and AI in Games* (under revision).
- G. Amador, A. Gomes. Best Neighbor First Search: A Fast and Simple Pathfinding Algorithm. *International Journal of Computer Games Technology* (submitted).
- M. Adaixo, G. Amador, A. Gomes. Algoritmo de Dijkstra com Mapa de Influência de Atratores e Repulsores. In Proceedings of the 2015 Portuguese Conference

of Sciences and Arts of Video Games (Videojogos'15), November 12-13, Coimbra, Portugal, 2015.

- G. Amador, A. Gomes. JOT: A Modular Multi-purpose Minimalistic Massively Multiplayer Online Game Engine. In Proceedings of the 2016 Portuguese Conference of Sciences and Arts of Video Games (Videojogos'16), November 24-25, Covilhã, Portugal, 2016.

Refira-se que os capítulos desta dissertação são baseados nas versões escritas destes artigos.

Organização da Dissertação

Esta tese encontra-se organizada da seguinte forma:

- O ***Capítulo 1*** introduz as questões referentes ao planeamento de movimento no contexto de jogos de vídeo. Esse capítulo introduz os problemas que se pretendem resolver, a sua motivação e as razões que justificam sua importância.
- O ***Capítulo 2*** faz uma análise crítica da literatura mais significativa sobre o planeamento de movimento em jogos de vídeo. Faz ainda uma análise comparativa entre diferentes soluções de planeamento de movimento, nomeadamente campos de influência e algoritmos de descoberta de caminhos.
- O ***Capítulo 3*** descreve uma solução para integrar algoritmos de descoberta de caminhos A* e mapas de influência, que são governados por novas funções de custo que combinam as funções de custo tradicionais com funções de influência. Estes algoritmos são aqui designados por algoritmos de descoberta de caminhos baseados em influência.
- O ***Capítulo 4*** descreve como as funções de custo tradicionais baseadas em distância podem ser substituídas por funções de custo exclusivamente baseadas em influência, daí resultando uma nova categoria de algoritmos, chamados algoritmos de descoberta de caminhos centrados em influência.
- O ***Capítulo 5*** introduz dois novos algoritmos de descoberta de caminhos que não são fundíveis com campos de influência. O primeiro designa-se por algoritmo de pesquisa pelo melhor vizinho primeiro ("best-neighbor first search", do inglês, ou apenas BNFS), ao passo que o segundo se designa por algoritmo de pesquisa aparada pelo melhor vizinho primeiro ("trimmed best-neighbor first search", do inglês, ou apenas TBNFS).
- O ***Capítulo 6*** conclui a dissertação e apresenta alguns direções para trabalhos futuros.

Audiência Alvo

Esta dissertação enquadra-se no âmbito do planeamento de movimento, um tópico importante em inteligência artificial para jogos de vídeo. Assim, esta tese é de interesse para programadores e investigadores da indústria de jogos, bem como para académicos interessados em algoritmos de pesquisa em grafos, algoritmos de descoberta de caminhos em jogos, e outros temas de investigação relacionados.

Contents

List of Figures

List of Algorithms

List of Abbreviations

AI	Artificial Intelligence
BestFS	Best-First Search
BFS	Breath-First Search
BNFS	Best Neighbor First Search
CSF	Cost-So-Far
DFS	Depth-First Search
FS	Fringe Search
IM	Influence map
NPC	Non-Player Character
OS	Optimistic Search
PF	Potential Field
SS	Skeptical Search
TBNFS	Trimmed Best Neighbor First Search
UBI	Universidade da Beira Interior

Chapter 1

Introduction

The big picture of this thesis work has to do with the development of techniques and algorithms to perform motion planning in the context of video games. Specifically, we intend to show that it is feasible to integrate influence fields with pathfinding, and simultaneously to rid off the local extremum problem tied to influence maps and the lack of path adaptivity linked to pathfinders. A* [HNR68], Dijkstra's [Dij59], and best-first search [Poh70b] pathfinders will be the main testing pathfinders. It will also be shown that not all pathfinders can be reformulated in terms of influence maps; for example, one of such pathfinders is described in the thesis and is called Trimmed best-neighbor first search (TBNFS), which is a near optimal pathfinder. Moreover, it will be demonstrated that it is possible to design pathfinders exclusively based on influence costs, rather than distance costs.

1.1 Motion Planning in Games

Graph searching is an important topic in artificial intelligence [PM10]. Graph searching is known as motion planning in robotics and games [LaV06], and has to do with finding a path in a graph that describes the passable node locations of the game world. Recall that artificial intelligence is a branch of computer science knowledge that was first defined by John McCarty in 1956 as "the science and engineering of making intelligent machines" [McC07], although there are alternative definitions in the literature [RN03].

In games, motion planning is used to endow non-player characters (NPCs) with an intelligent-like behavior [BS04]; specifically, motion planning deals with finding paths to move NPCs from one place to another in a virtual game world. Hereafter, path planning will always be discussed in the context of video games, so motion planning is divided into two majors techniques: *path-finding* and *influence fields* [Rab02, Paa08].

1.1.1 Pathfinders

Pathfinding plays an important role in many knowledge areas, including video games [YS08] [VX09], communication network routing [TS01] [TPA12], robotics path planning [Ark86] [LS90], and global positioning system (GPS) navigation systems [SSP08]. A pathfinder refers to an algorithm that finds a route (if one exists) between a start and goal node within a weighted search graph, and this means that a pathfinder is complete.

Also, most pathfinders ensure that the found path is the lowest in cost, that is, it is optimal. Multiple optimal paths may exist, and this means that distinct optimal pathfinders may find different paths, but they still are optimal paths. The search graph is a discrete structure composed of nodes connected by edges. In games, these nodes are related to Cartesian coordinates of the game world. If an edge has an associated value (i.e. cost or weight), the search graph is weighted. If edges can solely be traversed in a single direction, the graph is said to be directed. Nevertheless, let us say that this thesis solely deals with positive weighted undirected graphs.

The most employed pathfinder in games is A* and its variants, namely, Dijkstra's algorithm, best-first search, and fringe search. A*, Dijkstra, and fringe search are complete and optimal pathfinders, while the best-first search is solely complete. But, these algorithms suffer from several limitations:

1. ***Path Adaptivity***. They cannot handle search graph changes during a path search, that is, they are not *adaptive pathfinders*. Recall that such changes include: *node removal* (e.g. a bridge in the game world is removed as a consequence of its destruction); *node or edge addition* (e.g. a crossing that connects two roads is added). In fact, there are a few adaptive pathfinders, namely, lifelong planning A* (LPA*) [KLF04] and D* (also known as dynamic A*) and its variants [Ste94, Ste95, KL05]. However, when search graph changes occur, they require more memory storage, I/O operations, and additional node cost recalculations that end up significantly decreasing their time performance. Consequently, adaptive pathfinders are not used in games. Instead, when such changes occur, a path is recalculated from the start node to the goal node once again [SKY08].
2. ***Time Performance***. They work off-line [Poh73, Kor85, GMN94, Ree99, BEHS05, DSC08, TR10, HBUK12, BB12], i.e., they are not included in the game loop of events because their processing times noticeably decrease the frame rate.
3. ***Path Smoothing***. They suffer from the *path smoothing* problem because they produce piecewise linear paths, which result in jagged aesthetically unappealing paths [Mar02, Ger06, ASK15]. This problem can be addressed by replacing the found path by a curved path [BMS04] generated from splines. However, unless we use interpolating splines (e.g., interpolating cubic spline), this solution is computationally expensive because using approximation splines (e.g., Bézier splines) on a grid introduces more computations to avoid collisions between the NPC moving along the curved path and obstacles in scene [Rab00].

In the research work behind this thesis, we have mainly addressed the first two problems listed above, path adaptivity and time performance. This work was accomplished using influence fields. As seen throughout the thesis, the problem of path smoothing can also be solved using influence maps because influence maps may be defined by smooth (or differentiable) functions, with the advantage of allowing NPCs inherently avoid obstacles.

1.1.2 Influence Fields

When using an influence field in motion planning, NPCs are steered to avoid collisions with obstacles, which mimics human-like movement, with one translational step performed per game loop update. Therefore, there is no start-to-goal path planning, but instead a convergence process towards the goal. A field associates each point in game space and time with a quantity: a vector for *potential fields* or *flow fields*, and a scalar value for *influence maps*.

Potential fields mimic the behavior of magnetic material moving within a magnetic field (defined mathematically by a vector field), which results from summing up subsidiary potential fields. Each subsidiary field flowing from a point is referred to as a propagator. A propagator delivers a potential field that gradually fades to zero [HJ08b]. Depending on its charge, a propagator can either be attractive (i.e. objects move towards it) or repulsive (i.e. objects move away from it). In games, propagators may be associated to static (e.g. building door) and dynamic game entities (e.g. bot) that move around the game world.

An influence map is mathematically defined by a scalar field. Its applications go beyond the use of potential fields in motion planning because an influence map can also be used for spatial and temporal reasoning, particularly in the analysis of terrain, as well as understanding events that occur over time [SJ12, OSU+13, LCCFL13].

In general, influence fields have various advantages over pathfinders, namely:

1. They work in real-time for a single agent.
2. They work in real-time for multiple agents. Recall that pathfinding only works for a single agent and is not a real-time motion planning technique.
3. They are adaptive because they can handle changes to the game world.

In spite of their versatility, influence fields suffer from two major limitations relative to pathfinders:

1. They are not optimal, which is a fundamental feature to set the difficulty in games; for example, in a racing game pathfinders may be set to calculate several routes, some more and some less optimal, giving more or less margin to win to a player.
2. They suffer from a non-termination problem (i.e. not complete) as consequence of the local extremum problem. This problem takes the form of a local minimum problem when a game agent moves in the direction of decreasing values of the field, and a local maximum problem when a game agent moves in the direction of increasing values of the field. When an agent becomes trapped at a local extremum of an influence map, it is no longer able to leave [Goo00], as there is no place in its vicinity to move. In fact, there is no neighbor with a higher field value

(when converging to increasing field values) or lower field value (when converging to decreasing field values) to move. In potential fields, this phenomenon occurs at point(s) where the total sum of vector forces reaches zero.

1.2 Thesis Statement

To achieve adaptive pathfinders without altering the search graph during a search, we follow the leading idea that it is possible to integrate influence fields with pathfinders, and consequently to address the problems listed above for pathfinders and influence fields.

Thus, we can put forward the thesis statement as follows:

> *It is possible to develop novel motion planners capable of merging pathfinders with influence fields to address the problems of local extremum, to reduce memory and time consumption relative to canonical pathfinders, to handle search graph changes during path search, and to mimic human-like movement*.

As seen somewhere in this dissertation, it is even feasible to design a pathfinder exclusively based on influence maps, using influence costs instead of distance costs.

1.3 Research Methodology

To demonstrate the validity of the thesis statement, we adopted the following methodology:

Dataset of game maps. We decided to use the HOG2 dataset of game maps available at `http://www.movingai.com/benchmarks/` [Stu12]. In particular, we used ten maps of Warcraft 3 (outdoor scenes) and ten maps of Dragon Age: Origins (indoor scenes). These maps are structured into 2D grids of square cells. Each map is then encoded as a search graph, whose nodes correspond to passable cells and edges denote their adjacency relationships between neighboring cells. Note that white (floor) or green-blue (swamp) cells correspond to passable cells, while the remaining cells correspond to obstacles; these non-passable cells are not nodes of the search graph. HOG2 dataset is thus the basis of our work.

Merging pathfinders with influence fields. This was a major challenge of this doctoral work. Here, we had to find answers for the following questions: Is any pathfinder mergeable with influence fields generated by attractors and repellers? What makes a pathfinder mergeable with an influence field? How does such a merge translate into a new cost function? Such a merging only works well in pathfinders that choose the next

node as the lowest cost node in the open queue. Thus, our merging technique works well only on pathfinders like A*, Dijkstra, and best-first search. But, it fails in other pathfinders such as fringe search because it performs iterative deepening or a depth-first search instead of breadth-first search. Note that a pathfinder mergeable with an influence field requires the reformulation of the cost function, which should combine distance and influence values.

Solving the local extremum problem of influence fields. As known, influence fields suffer from the problem of local extremum. At the beginning of this research work, we believed that merging influence fields with pathfinding would solve such a problem. In fact, any pathfinder always forces to move forward toward the goal node. Consequently, even though an influence field's local extremum is found, a given NPC does not get trapped, that is, it continues moving by the action of the pathfinder.

Reducing memory and time consumption relative to canonical pathfinders. We also believed that using attractors to follow through toward the goal and repellers to get away from obstacles and undesirable regions would reduce the search space of our influence-aware pathfinders relative to canonical pathfinders (e.g., Dijkstra's, A*, best-first search). In addition to such canonical pathfinders, memory and time benchmarking also included the following pathfinders: static weighted A* [Poh70a], dynamic weighted A* [Poh73], optimistic search [TR08], and skeptical search [TR10]. The placement of attractors and repellers was made manually for simplicity; it is feasible to do so in an automated manner using, for example, the minimum spanning tree of the game graph.

Path adaptivity. Path adaptivity is controlled by the placement, removal, and motion of attractors and repellers in the game map. For example, when a street is destroyed, its passable nodes turn into impassable nodes, so if these nodes were part of a path between two locations, the path has to be reconstructed locally. This can be done by placing repellers in both extremities of the street to get away from such street, avoiding so the re-computation of the entire path from the start node to goal node.

Path smoothness. The discretization of the game map through a grid of cells makes any path looking jagged, even when one uses diagonals (i.e. using an 8-neigborhood to pick up the next node of a path). To endow the human-like movement to an NPC, we have to smooth the jagged path, making it a curved path [BMS04]. The typical solution to this problem is using an approximating spline (e.g., a Bézier spline), but this geometric solution does not guarantee the path does not collide with obstacles in the scene [Rab00]. Furthermore, this solution is computationally expensive. Our solution consists in using an interpolating cubic spline [dB78], which does not suffer from Runge's phenomenon either [Run01]. Besides, in the case of an expected obstacle collision, we can place a repeller at such obstacle to slightly deviate the path automatically. But, if there is a need to eliminate the small oscillations of the cubic interpolation spline when it turns right or left, our solution is then the piecewise cubic Hermite interpolating polynomial [FC80].

1.4 Research Plan and Timeline

In the research work that has led to this dissertation we have been guided by the following schedule:

- A comprehensive study of the literature on pathfinding and influence fields in games. This study has led to the write-up of Chapter 2.

- The implementation of an extensible, modular testing framework for pathfinders. Its features include: generation of mazes via Kruskal's or Prim's algorithms; loading of HOG2 game maps from the HOG2 repository at `http://movingai.com/benchmarks/`; visual inspection for a single algorithm at a time; output of statistical data (e.g. time and memory occupancy) for a set of algorithms for several maps and/or mazes. This framework was used to generate the results listed in Chapters 3 to 5.

- The implementation and testing of our first technique that integrates fields with pathfinders, which is described in Chapter 3. At first, the idea was to integrate fields with pathfinders by simply summing influence values to distance-based cost function values. The results published in [AAG15] were not satisfactory, because such influence-based pathfinder only worked with Dijkstra's algorithm. Finally, an improved version of our technique was designed and implemented to ensure that the moving NPC was able to flee from repellers and to pass through attractors toward the goal.

- The implementation and testing of our second technique that integrates fields with pathfinders are detailed in Chapter 4. As for the first technique (Chapter 3), we were able to re-design and re-implement the A*, Dijkstra, and best-first search with influence maps, but this time without using distance costs. However, it was not possible to do the same for fringe search. In fact, changing the node cost to manipulate its priority in the evaluation of the next node only applies to breadth-first search based algorithms such as A*, Dijkstra, and best-first search, but not to fringe search.

- Nevertheless, the unsuccessful re-design and re-implementation of the fringe search with influence costs led to the algorithm detailed in Chapter 5, called the best neighbor first search (BNFS). During the implementation of BNFS, we realized that its optimality could be improved, which originated a novel algorithm (TBNFS) that outperforms A*, Dijkstra, and best-first search, as well as fringe search.

- The write-up of this Ph.D. dissertation.

1.5 Contributions

The first major contribution of this thesis lies in the integration of canonical pathfinders (i.e. A*, Dijkstra, and best-first search) with influence maps, which simultaneously solve the adaptivity problem of canonical pathfinders, and both local extremum and non-completeness problems of influence fields. In fact, the influence-aware pathfinders described in Chapters 3 and 4 point to the constitution of a new family of adaptive pathfinders that can be used in games, which may be eventually extended to a novel family of multi-agent pathfinders.

The second major contribution of this thesis has to do with the introduction of two novel sub-optimal pathfinders, namely best neighbor first search (BNFS) and trimmed best neighbor first search (TBNFS), which often are less memory and time consuming than A* pathfinders. Specifically, TBNFS is closer to optimal than the best-first search algorithm employed in games.

1.6 Publications

This thesis work resulted in several papers, namely:

- G. Amador, A. Gomes. Influence Field-Based A* Algorithms. *IEEE Transactions on Computational Intelligence and AI in Games* (under revision).
- G. Amador, A. Gomes. Influence-based Cost Function A* Algorithms. *IEEE Transactions on Computational Intelligence and AI in Games* (under revision).
- G. Amador, A. Gomes. Best Neighbor First Search: A Fast and Simple Path-finding Algorithm. *International Journal of Computer Games Technology* (submitted).
- M. Adaixo, G. Amador, A. Gomes. Algoritmo de Dijkstra com Mapa de Influência de Atratores e Repulsores. In Proceedings of the 2015 Portuguese Conference of Sciences and Arts of the Video Games (Videojogos'15), November 12-13, Coimbra, 2015.
- G. Amador, A. Gomes. JOT: A Modular Multi-purpose Minimalistic Massively Multi-player Online Game Engine. In Proceedings of the 2016 Portuguese Conference of Sciences and Arts of the Video Games (Videojogos'16), November 24-25, Covilhã, 2016.

It is worthy noting that the chapters of this dissertation are strongly based on these papers.

1.7 Organization of the Thesis

This thesis is organized as follows:

- ***Chapter 1*** is the current chapter. It introduces the issues of motion planning in the context of video games. This chapter lays out the addressed problem, its motivation, and the reasons that justify its importance.
- ***Chapter 2*** provides a critical review of the most significant literature about motion planning for video games. It provides an analysis and comparison between different solutions found in the motion planning literature, namely influence fields, and pathfinders.
- ***Chapter 3*** describes how to integrate A* pathfinders with influence maps, which are governed by novel cost functions that combine the traditional cost functions with influence functions. We call them influence-based pathfinders.
- ***Chapter 4*** details how distance-based cost functions of A* pathfinders can be replaced by influence-based cost functions. As a result, we have a new class of pathfinders, called influence-centric pathfinders.
- ***Chapter 5*** introduces two novel pathfinders that seemingly cannot be combined with influence maps. They are called best neighbor first search (BNFS) and trimmed best neighbor first search (TBNFS).
- ***Chapter 6*** concludes the thesis and presents points for further improvement in the future.

1.8 Target Audience

This dissertation lies in the scope of motion planning, an important topic in artificial intelligence for computer games. Thus, this thesis is of interest to programmers and researchers in the game industry, as well as academics interested in graph search algorithms, pathfinding, and related research topics.

Chapter 2

Motion Planning For Games: a Survey

The gaming industry is a billion dollar market [APS08], whose knowledge has application beyond games, e.g., gamification [Ass15]. Artificial Intelligence techniques are common in video games, among which we count on tactical planning, natural language processing and learning, and motion planning [LMP+13, Rab13, Rab15]. This chapter addresses motion planning algorithms. In particular, we focus on the benchmarking pathfinders used in this thesis, namely Dijkstra's search, A*, best-first search, fringe search, and a few general-purpose suboptimal bounded searches.

2.1 Introduction

Artificial intelligence is branch of computer science knowledge that was first defined by John McCarty in 1956 as "the science and engineering of making intelligent machines" [Var12], though other authors provide us with alternative definitions [RN03]. Motion planning is a term that we find in different research topics and knowledge areas, including control theory [MKH99], robotics [YQS+16], games [Nar04], genetic algorithms [LLM07], and, more generally, artificial intelligence [LaV06]. In games, motion planning is used in the context of artificial intelligence to make non-player characters (NPCs) appear human-like when they move in the game world [BS04]. More specifically, motion planning in games deals with finding routes/paths to move NPCs from one place to another in a virtual game world. In games, motion planning is divided in two majors techniques: *path-finding* and *influence fields* [Rab02] [Paa08].

Historically, there is no clear reference to when for the first time were either fields or pathfinders employed in video games, but there are some pathfinders that clearly precede late 1960's first commercial video games, as it is the case of the depth-first search (DFS) [Mac96], breadth-first search (BFS) [Moo57], and Dijkstra's algorithm [Dij59]. That is to say that, some pathfinders were applied to video games years after those pathfinders were introduced into scientific literature. As shown in Fig. 2.1, influence fields (in particular, influence maps) were the first motion planning technique employed into games; specifically, the concept of influence map was used for the first time in the GO game and is due to Zobrist [Zob69]. However, it was solely used to choose the next tactical decision or GO move; that is, it was not used for motion planning of NPCs or game agents.

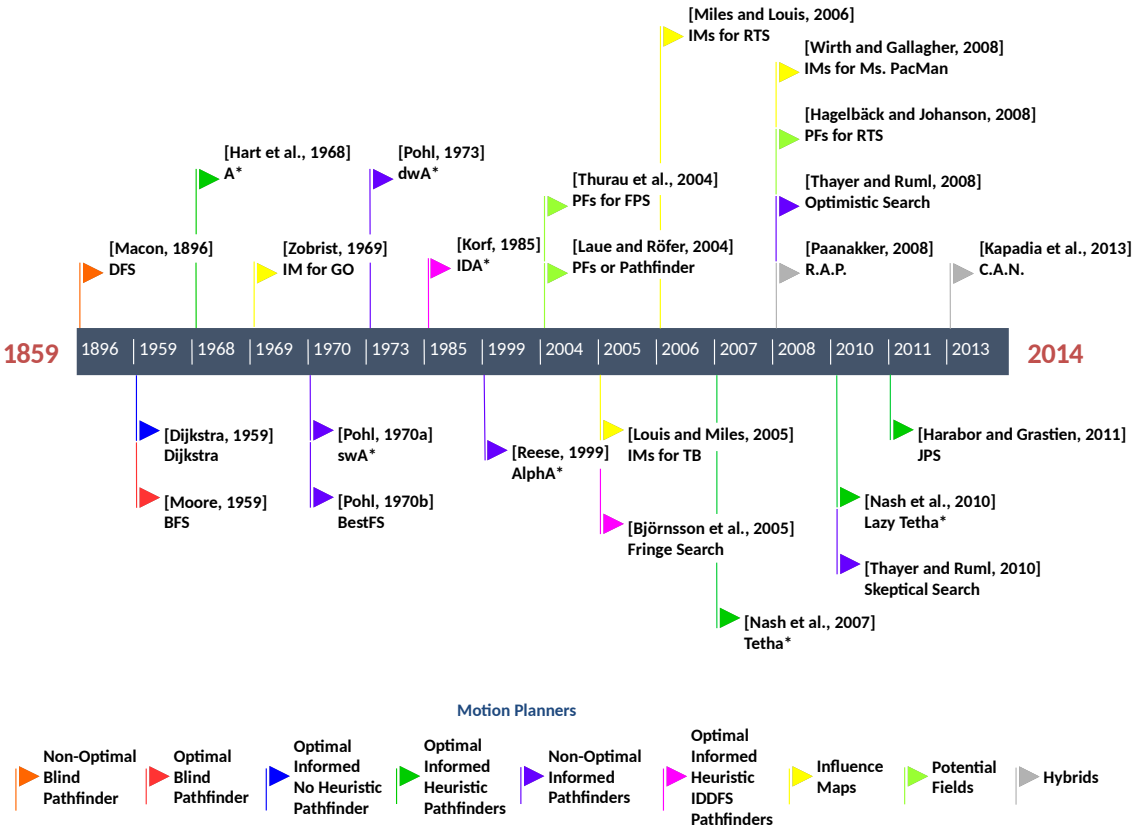


Figure 2.1: Motion planning timeline.

2.2 Motion Planning Timeline

The timeline of the most representative motion planners is shown in Fig. 2.1. However, in this section we only address those that were implemented or are pertinent to the research work behind this thesis.

Dijkstra [Dij59]. In 1959, Dijkstra proposed the first solution for the traveling salesman problem using graph theory, which consists in finding the shortest path between two vertices (or nodes) of a search graph. Dijkstra's algorithm is the first optimal and complete pathfinder.

A* [HNR68]. In 1968, Hart et al. proposed A* with the purpose of reducing Dijkstra's algorithm search graph expansion by means of a *heuristic function* or simply *heuristic*. This means that A* outperforms Dijkstra's algorithm in time performance and memory space consumption. The heuristic is an estimate of the cost from the current node to the goal node of the graph. To ensure A* optimality, this estimate must always be below the real cost of traversing the minimal path between the current node and the goal node. Note that Dijkstra can be seen as a particular case of A* with the heuristic function set to zero. In short, A* is a complete and optimal pathfinder under certain conditions.

Best-first search [Poh70b]. In 1970, Pohl proposed the best-first search and the first

general-purpose sub-optimal bounded pathfinder called ***static weighted A**** [Poh70a]. Both are complete but non-optimal pathfinders. Best-first search is in fact the particular case of A* when solely the heuristic function is used. Static weighted A* pathfinder is intended to improve A* convergence to the goal by reducing the number of evaluated graph nodes. In fact, both best-first search and static weighted A* often are faster and consume less memory than A*, at the expense of not being optimal pathfinders. That is, either best-first search or static weighted A* pathfinder is unable to ensure A* optimality.

Dynamic weighted A* [Poh73]. In 1973, Pohl introduced another general-purpose bounded sub-optimal pathfinder, called dynamic weighted A*. The core idea of this algorithm is to increasingly reduce graph expansion as the search gets closer to the goal node. Like the static weighted A*, the dynamic weighted A* is also complete but non-optimal.

IDA* [Kor85]. In 1985, Korf introduced the iterative deepening A* (IDA*), with the intent of making a more efficient memory usage than A*. For that purpose, IDA* discards both the open and closed sets used in A*, as it performs a depth-first search (DFS), not a breadth-first search (BFS), from the start node until a certain node level (threshold). However, the gain in memory occupancy that IDA* algorithm has over A* results in a significant increase in computations that cripples the time performance to a point that makes it a barely viable alternative to A*. Even so, IDA* is a complete and optimal pathfinder.

D* [Ste94]. In 1994, Stentz introduced the first adaptive pathfinder, called D*, which is also known as dynamic A* [Ste94]. Adaptive pathfinders are complete but not necessarily optimal pathfinders. Contrary to former pathfinders, they allow to handle changes to the graph while searching for a path between two locations; for example, a bridge in the game world is removed as a consequence of its destruction (node removal). Unfortunately, adaptive pathfinders require additional data to handle node cost changes. Thus, adaptive pathfinders require more memory storage and I/O operations and, consequently, additional node cost recalculations. Hence, the predominantly employed pathfinders in video games are Dijkstra [Dij59] and A* [HNR68]. This means that usually pathfinders cannot deal with changes to the search graph (i.e., removal/addition of nodes from/to the graph) without recalculating from the start node to the goal node [Ste94, Ste95, KLF04, KL05, SKY08].

AlphA* [Ree99]. In 1999, Reese introduced AlphA*, yet another general-purpose bounded sub-optimal pathfinder, which prioritizes recently expanded nodes to promote depth-first exploration. But, as usual for general-purpose sub-optimal bounded searches, AlphA* is also complete but non-optimal.

Fringe search [BEHS05]. In 2005, Björnsson et al. proposed the fringe search, a middle term between A* and IDA*. Fringe search is an improvement on IDA*, because instead of recalculating if a path exists from the start node to a certain graph expansion level threshold, fringe search stores the last leaf nodes evaluated (to see if any is the goal), referred to as fringe nodes, and continues to expand the graph until the goal is

reached or the search graph as been completely expanded/explored. Consequently, fringe search is faster than A* and IDA* algorithms and has less memory requirements than IDA*. Nevertheless, possibly due to being relatively new (2005) when compared to A* (1968), fringe search is more sparsely employed in games. In short, fringe search is a complete and optimal pathfinder that becomes a viable alternative to IDA*, because it outperforms both A* and IDA*.

Optimistic search [TR08] [TR10]. In 2008 and 2010, Thayer and Ruml presented yet two novel general purpose suboptimal bounded searches, called optimistic search and ***skeptical search***. As usual for general-purpose sub-optimal bounded searches, these two pathfinders attempt improve convergence, yet they also attempt to improve the optimality of the found path as much as possible.

2.3 Canonical Pathfinders

This section describes the canonical pathfinders used in this thesis, namely: Dijkstra's, A*, best-first search, and fringe search.

2.3.1 Dijkstra

Dijkstra's leading idea is to choose the next node to evaluate the one that has the lowest cost-so-far (CSF) to the start node. That is, Dijkstra's uses a *backward-looking strategy* in the sense that the cost of a node is calculated in relation to the start node; no cost is considered in relation to the goal node. The search space grows outwards from the start node according to a breadth-first strategy, what means that the collection of neighbor nodes is an ordered priority queue (first-in-first-out principle). The queue is often referred to as open set. This means that the CSF value associated to a node may be altered if a lower path is found, i.e., nodes can be revisited/re-evaluated. The algorithm only stops, when all possible paths have been evaluated, as shown in Alg. 1.

Dijkstra's algorithm initialization consists in setting the start node's parent reference to null and its G cost to 0 (line 1), followed by adding the start node the open set (line 2). In each iteration, while the open set is not empty, the algorithm removes first node —assuming the open set is an ordered priority queue whose first node is always the one with the lowest cost so far value— and sets it to the current node N (lines 3-4). Afterwards, all nodes neighboring the current nodes are expanded/evaluated by CSF value, so that if a node has a better (lower) CSF, one adds it to the open set (if not already there), and if it is stored in the closed set, one removes it for reevaluation (lines 5-19). When the CSF of a node is updated, its parent's CSF is also altered. When the open set is empty, a path (if one exists) is surely to be found when reconstructing a path (line 20), starting from the goal node back to the parent of each node until the start node is reached (there is a path) or a node has a parent still undefined (there is

Algorithm 1 Dijkstra's Search Algorithm

```
 1: Clear start node's parent and G cost.
 2: Add start node to open set
 3: while open set is not empty do
 4:    n = poll best node from open set
 5:    Add n to closed set
 6:    for all Neighbor nodes N of n do
 7:       Calculate new G cost
 8:       if N is unexplored then
 9:          Set N's G cost to new G cost
10:          Set N's Parent to n
11:          Add N to open set
12:       else if N is on the open set and N's G cost is greater than new G cost then
13:          Set N's G cost to new G cost
14:          Set N's Parent to n
15:       else if N is on the closed set and N's G cost is greater than new G then
16:          Set N's G cost to new G cost
17:          Set N's Parent to n
18:          Remove N from closed set
19:          Add N to open set
20: Reconstruct path
```

no path to be found). The memory costs of Dijkstra's algorithm accounts for both the data contained in each node in the graph and the memory costs of the open and closed sets.

There are two small optimizations that can be performed on Dijkstra's algorithm:

1. In each iteration, since nodes are explored based on CSF relative to the start node, a less costly path to a closed node is highly unlikely to be found [Ang12]. Therefore, neighbor nodes already in the closed set may be discarded from cost reevaluation, i.e., lines 14 to line 18 from Alg. 1 can be discarded.

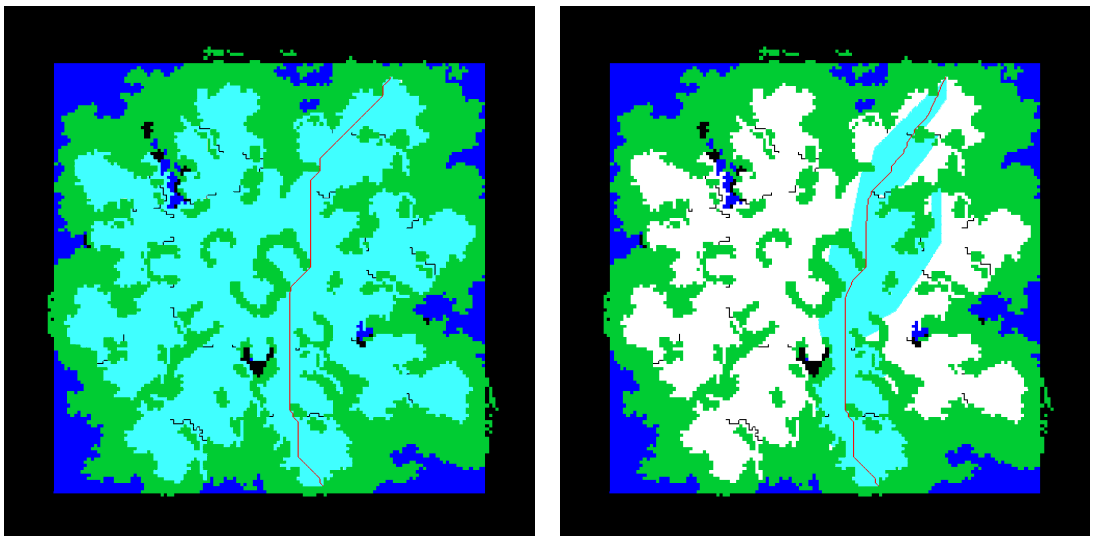
2. In games, a point-to-point search version of Disjktra's suffices, i.e., the algorithm stops when one of the explored nodes is the goal node.

Dijkstra's algorithm with such optimizations is described in Alg. 2.

Fig. 2.2(a) shows a path found with the former improved version of the Dijkstra algorithm, in the HOG2 crucible map of Warcraft 3. Note that the found path is in red, expanded nodes are in cyan. HOG2 maps are structured into grids of squares (called nodes); passable nodes are in white, while impassable nodes are in black (out of bounds), green (vegetation), and blue (water). See [Stu12] for further details about HOG2 maps.

Algorithm 2 Point-to-Point Dijkstra's Search Algorithm

```
1:  Clear start node's parent and G cost.
2:  Add start node to open set
3:  while open set is not empty do
4:     n = poll best node from open set
5:     Add n to closed set
6:     if n is goal then
7:        Reconstruct path
8:     for all Neighbor nodes N of n do
9:        Calculate new G cost
10:       if N is unexplored then
11:          Set N's G cost to new G cost
12:          Set N's Parent to n
13:          Add N to open set
14:       else if N is on the open set and N's G cost is greater than new G cost then
15:          Set N's G cost to new G cost
16:          Set N's Parent to n
17: No Path
```



(a) (b)

Figure 2.2: Paths (in red) found in HOG2 crucible map of Warcraft 3 through: (a) point-to-point Dijkstra's algorithm; (b) A*.

2.3.2 A*

Historically, Dijkstra's algorithm was the first optimal and complete pathfinder introduced in the literature to solve the shortest path problem. A* is a follow-up of Dijkstra's. A* is the dominant algorithm for path-finding in computer/video games, mostly because it requires to visit less nodes in the search graph than Dijkstra's algorithm to find the best path (if one exists).

Its leading idea is what is the CSF value to get to a node plus the likelihood of it to converge faster to the goal. The likelihood of a node converging to the goal is based on an estimate of its cost to the goal, referred to as heuristic in the literature; for example,

the Euclidean distance from node to goal is quite often used as heuristic. If the heuristic value is always zero, A* behaves as the Dijkstra's algorithm, so that Dijkstra's algorithm can be seen as a particular case of A*.

The use of a heuristic implies that A* is solely optimal if the employed heuristic is admissible, i.e., if the heuristic value is always lower than the real cost from either graph node to the goal node [BM83]. Also, by resorting to an heuristic, A* will behave as an hybrid pathfinder that performs a goal oriented depth-first search (DFS) and performs a breadth-first search (BFS) to overcome obstacles, reverting to a depth-first search when an obstacle is overcomed, e.g., when a dead end is found.

The heuristic serves two purposes. First, it allows us to steer the search to achieve the mentioned faster convergence. Second, it allows us to reduce the amount of nodes required to traverse when compared to Disjktra's algorithm. Roughly, less than 50% of nodes required for Disjktra's algorithm are evaluated in A*, yet with high increased computation load, as a result from the calculation of the heuristic for each neighbor of a node being evaluated, and memory requirements, as the value of heuristic has to be also stored.

Both Dijkstra and A* resort to an open set that stores all nodes yet to be evaluated. For efficiency sake, the open set in either algorithm is often represented as an ordered priority queue, whose first value is the node with the lowest cost (i.e., in A* with the lowest CSF plus heuristic value). In spite of A* requiring additional computations, comparatively to Dijkstra's, due the the use of an heuristic function, as long as the appropriate data structure for the open set is used (i.e., ordered priority queue), A* will expectedly perform better in total time and memory requirements than Dijkstra's algorithm, i.e., there is more load per node but less nodes evaluated overall.

Algorithm 3 A* Search Algorithm

```
 1: Clear start node's parent and cost values ( F, G, H )
 2: Add start node to open set
 3: while open set is not empty do
 4:    n = poll best node from open set
 5:    Add n to closed set
 6:    if n is goal then
 7:       Reconstruct path
 8:    for all neighbor nodes N of n do
 9:       if N is unexplored then
10:          Calculate new G cost ( Current G + traversal cost )
11:          if N is not in open set or new G cost is lower than N's G cost then
12:             Set N's Parent to n
13:             Set N's G cost to new G cost
14:             Set N's F cost to N's G cost + Heuristic estimate from N to Goal
15:             if N is not in open set then
16:                Add N into open set
17: No Path
```

A* algorithm described in Alg.3 differs slightly from Alg. 2. First, its initialization differs

from Dijkstra's as it also requires to start by setting the start node values, F, G and H to zero, setting also its parent reference to null (line 1). Second, its lines 9 to 16 differ from their homologous in Dijkstra's, because A* alters the node cost if the CSF is cheaper than before, storing it in the open set based on the CSF plus the heuristic. Fig. 2.2(b) illustrates a path found (in red) with A*, in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.

2.3.3 Best-first search

Best-first search is the particular case of A* when the CSF value is always zero, as solely the heuristic function is employed. The leading idea of best-first search is to choose as next node to evaluate the one with the lowest estimate to reach the goal node. That is, it uses a forward-looking strategy. To put it in other words, the next node chosen to evaluate will be the node most likely to converge faster to the goal, i.e., with the lowest heuristic value.

Removing the CSF value from the cost function implies that best-first search performs an informed DFS. In fact, knowing the goal position and using such information to estimate the lowest cost next node to evaluate is an informed choice. Discarding the CSF value implies that best-first search in not guaranteed to be optimal, i.e., the heuristic ignores obstacles, which implies a zig-zag like movement and non-optimal traversal to overcome an obstacle. This is why best-first search is also referred to as a greedy algorithm, in the sense that it will take a route to overcome an obstacle not ensuring that a better alternative route exists.

However, best-first search often outperforms both A* and Dijkstra, and in some cases fringe search, both in time and memory requirements [DP85]. Regarding Dijkstra's algorithm, this behavior makes sense since a depth-first search (DFS) strategy is faster and has lower memory consumption than a breadth-first search (BFS) strategy. A* is beaten by best-first search for two main reasons. First, A* performs BFS when overcoming an obstacle thus consuming more time and taking more steps before returning to a DFS convergence to the goal. Second, best-first search does not use the CSF value and thus requires less calculations and memory read and write operations per iteration.

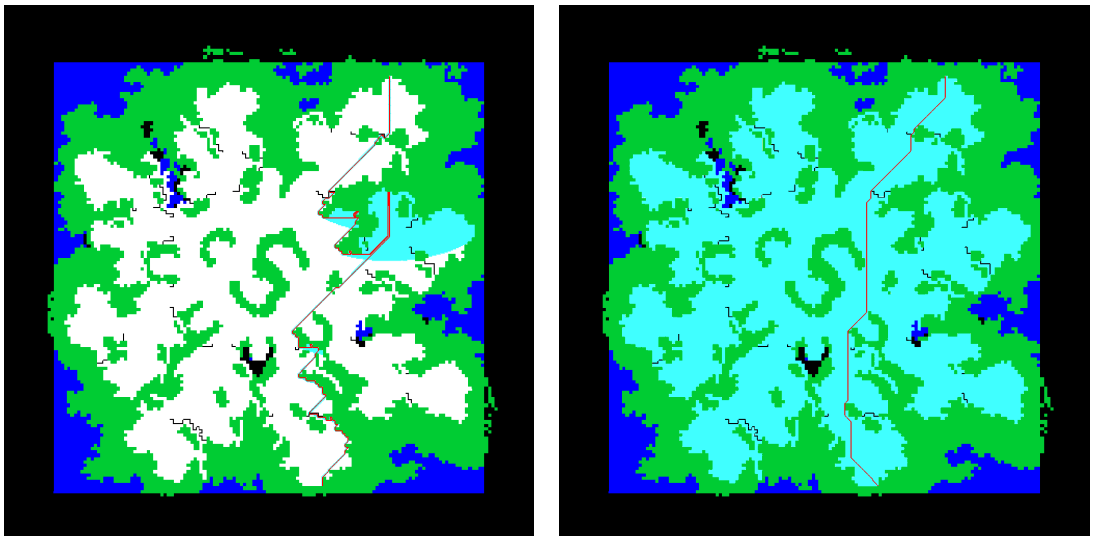
Finally, best-first search also beats fringe search, also an informed DFS, both in memory and time requirements for two reasons. First, less computation, as no CSF value is calculated. Second lower amount of memory read and write operations, as fringe search often requires an auxiliary copy of the open set in each iteration. Best-first search algorithm is detailed in Alg. 4.

The best-first search, like A*, works with an open set and a closed set (see Alg. 4). What distinguishes the best-first search from A* is that if a node is not within the closed set its cost and its parent are updated; otherwise they are the same, with the exception that best-first search solely resorts to the heuristic (lines 9 and 11). Fig. 2.3(a) illustrates a path found (in red) with best-first search in the Warcraft 3 thecrucible map, while the

Algorithm 4 Best-First Search

```
1:  push start node to open set
2:  while open set is not empty do
3:     n = poll best node from open set
4:     Add n to closed set
5:     if goal is found then
6:        Reconstruct path
7:     for all Neighbor N in n do
8:        if N is unexplored then
9:           Calculate new H cost.
10:          Set N's Parent to n
11:          Set N's F cost to N's Heuristic estimate from N to Goal
12:          if N is not in open set then
13:             Add N into open set
14: No Path
```

expanded nodes are in cyan.



(a) (b)

Figure 2.3: Paths (in red) found in HOG2 thecrucible map of Warcraft 3 through: (a) best-first search; (b) fringe search.

2.3.4 Fringe search

The leading idea of fringe search is to perform DFS, using as threshold the CSF value plus an heuristic metric as in A*, storing the frontier or fringe nodes whose cost is below of the current threshold. Similar to A*, fringe search looks backward and forward during the search, but they differ from one another in how the next node is chosen. Fringe search algorithm is detailed in Alg. 5.

Fringe search initialization consists in setting the threshold limit to 0 and found path variable to false (lines 1-3). Then, it iterates over the list $\mathbb{F}$ of fringe nodes while it is not empty and the path is not found (line 4) as follows:

Algorithm 5 Fringe search algorithm

```
 1: fringe 𝔽 = s
 2: cache C[start] = (0, null)
 3: found = false
 4: while (found is false) AND (𝔽 not empty) do
 5:    min_limit = ∞
 6:    for all N in 𝔽, from left to right do
 7:       (G, parent) = C[node]
 8:       F = G + H(node)
 9:       if F is greater than F_limit then
10:          min_limit = min(F, min_limit)
11:          Continue
12:       if node is goal then
13:          found = true
14:          break
15:       for all child in children(node), from left to right do
16:          child G = G + cost(node, child)
17:          if C[child] is not null then
18:             (G cached, parent) = C[child]
19:             if G child is greater than or equal to G cached then
20:                Continue
21:          if child in F then
22:             Remove child from 𝔽
23:          C[child] = (G child, node)
24:       Remove node from 𝔽
25:    F_limit = F_min
26: if reached goal is true then
27:    Reconstruct path
```

1. Set the minimum threshold to maximum value (line 5).
2. For all the nodes in $\mathbb{F}$, cache the node and set the new threshold (lines 6-7).
3. If the threshold is higher than the maximum threshold, it is clamped to the minimum value (lines 9-11).
4. If the goal is found, set found variable to true and break out of the loop (lines 12-14).
5. For all children of the examined node, calculate new threshold (lines 15-16).
6. If child in $\mathbb{F}$, remove child from $\mathbb{F}$ (lines 21-22).
7. Keep child in cache (line 23).
8. Remove examined node from $\mathbb{F}$ (line 24).
9. Update threshold limit to minimum threshold (line 25).
10. If the goal was reached, reconstruct path (lines 26-27).

Fig. 2.3(b) illustrates a path found (in red) with Fringe search, in the Warcraft 3 the-crucible map, while the expanded nodes are in cyan.

A* and Dijkstra's pathfinders are discrete approaches. Parallel or hierarchical variants of A* exist among other to deal with the strict requirements in memory and computation of video games.

Similarly to A* and Dijkstra's pathfinders, fringe search is also a discrete pathfinder. Fringe search can be also seen as a fastest variant of iterative deepening A* (IDA*). Recall that IDA* is a variant of A* that addresses the memory cost of A*, while using the same steered search principle of A* based on an heuristic function. In each iteration, IDA* traverses node branches stopping on nodes whose cost is above a given threshold or depth. If the goal node is found in one of the peripheral or border nodes, the best path is found; otherwise, a new larger threshold is attempted until either no node exists above the given threshold or a path is found. The problem with IDA* is that, for each threshold, if a path is not found all computations (node traversal) must be redone for the next threshold. This results in an algorithm that does not possesses the same memory cost of A*, at the expense of a prohibitive increase in computation. As a result, IDA* is often discarded as viable for games.

Fringe search, as an improved version of IDA*, reduces computation at the cost of further memory to store the search frontier for the end of each threshold branch or fringe. Therefore, if for a given threshold the goal node is not found, solely nodes from branches below the last fringe nodes must be evaluated. Thus, similar to IDA*; fringe search follows a many-fronts approach. In short, fringe search outperforms A* and Dijkstra' in time performance, but it requires to visit more nodes than A* and sometimes even more than Dijkstra's, yet in overall it evaluates faster more nodes, as fringe search often requires less memory than either A* or Dijkstra's because it only stores the list of fringe nodes.

2.4 General-Purpose Suboptimal Bounded Searches

One of the solutions to the memory consumption problem is to reinforce constraints to the length of (see [GMN94]) or even discard (see [Kor85, BEHS05]) the open and closed sets used by A*. Another solution is to alter the value of the cost function heuristic, in order to restrain graph expansion, resulting in a reduction of memory and time constraints of A* at the cost of optimality. This solution is what is performed by general-purpose bounded suboptimal searches, namely, static weighted A* [Poh70a], dynamic weigheted A* [Poh73, TR09], alphA* [Ree99], and optimistic and skeptical searches [TR08, TR10].

These bounded searches work by relaxing A* heuristic function admissibility criterion. While the admissibility criterion guarantees an optimal solution path, it also means that A* must examine all equally meritorious paths to find the optimal path. It is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion.

Often, we want to bound this relaxation so that we can guarantee that the solution path is no worse than (1 + ε) times the optimal solution path, i.e., that its ε admissible.

What differs in the general-purpose suboptimal bounded searches described below is how they change the cost function of A* to reinforce that a found path is ε admissible.

2.4.1 Static Weighted A*

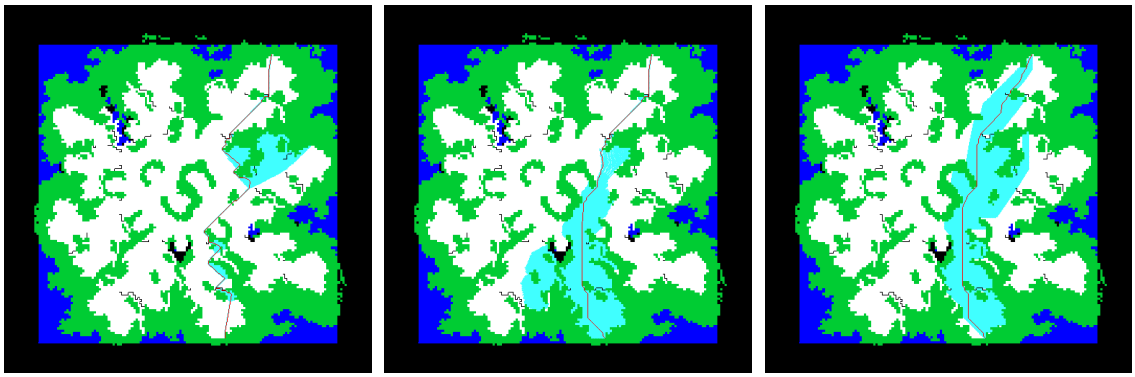
Static weighted A* modifies the cost function of A* as follows:

$$F(n) = G(n) + w \,.\, H(n) \tag{2.1}$$

where w is the weight, $G(n)$ refers to the cost of the current node to the start node, and $H(n)$ denotes the cost of the current node to the goal node. The heuristic $H(n)$ represents the likelihood of the current node converging faster to the goal, which is based on an estimate of the cost to the goal; for example, a possible heuristic is the Euclidean distance from the current node to goal node. The value of w is given by the following expression:

$$w = 1 + \varepsilon \tag{2.2}$$

where $\varepsilon > 1$, and ε is a constant set before performing path-finding; hence, the name 'static weighted A*'. Using ε results in fewer expanded nodes, i.e., a path is found whose cost at most can be ε times more than the cost of the optimal path that A* would find otherwise. Fig. 2.4(a) illustrates a path found (in red) with static weighted A* in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.



(a) (b) (c)

Figure 2.4: Paths (in red) found in HOG2 thecrucible map of Warcraft 3 through: (a) static weighted A*; (b) dynamic weighted A*; (c) alphA*.

2.4.2 Dynamic Weighted A*

In the dynamic weighted A*, the weight w associated to the cost function is dynamic rather than static. Static weighted A* results in a restriction of A* search during the

entire path search. Instead, dynamic weighted A* relaxes expansion in the beginning of the search, progressively constraining more and more the expansion of nodes in the way toward the goal node. Thus, it changes the weight dynamically during the path search; hence, the name 'dynamic weighted A*'.

The cost function of A* of the dynamic weighted A* is as follows:

$$F(n) = G(n) + (1 + \varepsilon \times w(n)) \, . \, H(n) \tag{2.3}$$

where $w(n)$ is given by the following expression:

$$w(n) = \begin{cases} 1 - \dfrac{d(n)}{N} & , d(n) \leq N \\ 0 & , \text{otherwise} \end{cases} \tag{2.4}$$

where, $d(n)$ is the depth of the search and N is the anticipated length of the solution path. Fig. 2.4(b) illustrates a path found (in red) with dynamic weighted A* in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.

2.4.3 AlphA*

AlphA* attempts to promote depth-first instead of the standard breadth-first performed by A* by prioritizing to recently expanded nodes. In order to do so, it uses the following cost function:

$$F(n) = (1 + \varepsilon \, . \, w(n)) \, . \, (G(n) + H(n)) \tag{2.5}$$

where w is given by the following expression:

$$w = \begin{cases} \lambda & , g(\pi(n)) \leq g(\tilde{n}) \\ \Lambda & , \text{otherwise} \end{cases} \tag{2.6}$$

where Λ is a constant, $\lambda \leq \Lambda$, $\pi(n)$ the parent of node n, $\tilde{n}$ is the latest expanded node, and the function g is either G or H. Fig. 2.4(c) illustrates a path found (in red) with alphA* in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.

2.4.4 Optimistic Search

Optimistic search uses arbitrary heuristics for at least a portion of its search. In other words, it runs dynamic weighted A* with a weight higher than the desired suboptimality bound, that is, to find an incumbent node. Its leading idea is to achieve better optimality than prior general-purpose suboptimal bounded searches. However, as any other general-purpose suboptimal bounded search, it aims to find solutions whose quality is within a fixed range of optimality. Optimistic search algorithm is described in Alg. 6.

Algorithm 6 Optimistic Search Algorithm

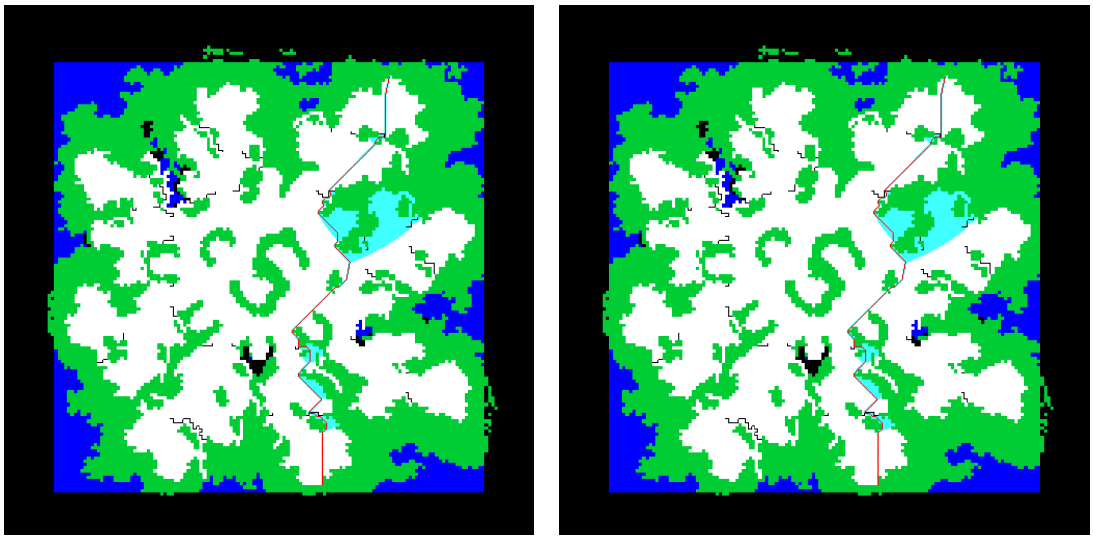
```
1: incumbent ← null
2: openset ← root
3: while incumbent = null and openset ≠ ∅ do
4:    remove n from openset with minimum F'(n) = G(n) + w . H(n)
5:    if n is a goal then
6:       incumbent ← n
7:    else
8:       expand n and insert children into openset
9: while openset ≠ ∅ do
10:   n_min ← n ∈ open with minimum F(n) = G(n) + H(n)
11:   n'_min ← n ∈ open with minimum F'(n) = G(n) + w . H(n)
12:   if b . F(n_min) ≥ G(incumbent) then
13:      return incumbent
14:   else if F'(n'_min) ≤ G(incumbent) then
15:      if n'_min is a goal then
16:         incumbent ← min(n'_min, incumbent)
17:      else
18:         remove n'_min from openset, expand it, and insert its children
19:   else
20:      remove n_min from openset, expand it and insert children into openset
21: return incumbent
```

This initial solution is determined using dynamic weighted A*, for a weight w higher than the bound b (lines1-8). Afterwards, additional nodes are expanded like in A* until a found solution can be proven to be within a desired suboptimality bound (lines 9-20), and if such said solution is returned. Specifically, it must be proved that the incumbent within the desired suboptimality (lines 12, 13, and 20). Also, if possible the incumbent should be improved (lines 14-16). Fig. 2.5(a) illustrates a path found (in red) with optimistic search in the Warcraft 3 thecrucible game map, while the expanded nodes are in cyan.

2.4.5 Skeptical Search

Skeptical search differs from optimistic search in the sense that it is skeptical about placing absolute trust in the base heuristic. This implies that skeptical search, comparatively to optimistic search, has no parameter tuning requirements and often performs better that optimistic search. This means that potential solutions are found faster, but tend to be of inferior quality (less suboptimal). Skeptical search algorithm is detailed in Alg. 7 for convenience sake.

Alg. 7 differs from Alg. 6 essentially by not using any inadmissible heuristic. Instead, the weighted admissible heuristic from the first phase of optimistic search is replaced with any learned heuristic (line 12). Fig. 2.5(b) shows a path found (in red) with skeptical search in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.



(a) (b)

Figure 2.5: Paths (in red) found in HOG2 thecrucible map of Warcraft 3 through: (a) optimistic search; (b) skeptical search. Both paths look the same because they both algorithms are based on static weighted A*.

Algorithm 7 Skeptical Search Algorithm

```
1:  incumbent ← null
2:  openset ← root
3:  while incumbent = null and openset ≠ ∅ do
4:      remove n from openset with minimum F̂'(n) = G(n) + w . Ĥ(n)
5:      if n is a goal then
6:          incumbent ← n
7:      else
8:          expand n and insert children into openset
9:  while openset ≠ ∅ do
10:     n_min ← n ∈ open with minimum F(n) = G(n) + H(n)
11:     n̂'_min ← n ∈ open with minimum F̂'(n) = G(n) + w . Ĥ(n)
12:     if w . F(n_min) ≥ G(incumbent) then
13:         return incumbent
14:     else if F̂'(n̂'_min) ≤ G(incumbent) then
15:         if n̂'_min is a goal then
16:             incumbent ← min(n̂'_min, incumbent)
17:         else
18:             remove n̂'_min from openset, expand it, and insert its children
19:     else
20:         remove n_min from openset, expand it and insert children into openset
21: return incumbent
```

2.5 Influence Fields

Unlike pathfinders, influence fields were not thought of to find a path between two locations, but solely to induce a steering motion on game entities that move around the environment. When using influence fields, there is not path planning from a start to a goal position. Instead, entities are steered in order to avoid collisions with obstacles while moving around the field, which results in an human-like movement of NPC's and

game agents with one step (translation) performed per game loop update. Fields refer essentially to a quantity that has a value for each point in game space and time. If these quantities are scalar values we refer to them as *influence maps*, while if instead these quantities are vectors we refer to them as *potential fields* or *flow fields*.

Artificial *potential fields* are a robotics technique [Kha85][Ark86][Ark87], that was latter introduced into games by the hand of Thurau et al. [TBS04]. The core idea is to emulate the movement of a magnetic material within an electromagnetic field, that results from summing up subsidiary potential fields. A propagator is the center of each subsidiary potential field, that delivers a potential field that gradually fades to zero [HJ08b]. There are two kinds of propagators: attractive, whose charge makes objects move towards it; repulsive, whose charge makes objects move away from it. Propagators may be tied to static and dynamic game entities (e.g., a building as a static entity and a bot as a dynamic entity that moves around) or not. With fields, motion planning, or more broadly decision-making, can be done for multiple entities simultaneously, as done for real-time strategy (RTS) games [HJ08a, HJ09b, HJ09a]. For example, when steering an entity by simply considering other entities and/or obstacles as repellers, a moving entity will naturally avoid any static or moving obstacle.

On the other hand, influence fields are the scalar version of potential fields, attained by replacing vectors by a weight/cost, e.g., the norm of a vector may be used to transform a potential field's vectors into an influence field's influences. Influence fields where first employed into games for the GO game [Zob69], where influence maps where used to chose the next tactical decision or GO move, i.e., not for motion planning of avatar(s). Influence fields, arose from the limitation of potential fields that solely allow motion planning of avatars. Thus, influence maps/fields can be used also for spatial and temporal reasoning, namely, analysis of terrain, as well as understanding events that occur over time [SJ12, OSU+13, LCCFL13].

For example, in a game there can be four layers of (i.e., four distinct) influence fields, specifically, for vision, hearing, smell, and distance. In order to take a decision, avatar(s) may use either individual layer or the sum of either combination of two or more layers. Therefore, a decision at night might be taken by an avatar solely based on hearing, while a decision during the day may instead be decided solely based on vision. In fact, a myriad of information may be considered by game agents in order to allow them to take strategic decision that strives on achieving an overall goal; for example, steel an enemy's briefcase and bring it back to avatar base. Such a strategic decision consists of a series of tactical decisions, that is, a number of small and precise actions; for example, where to go in order not to be detected by an enemy involves handling faction influence, town control influence, combat strength, visibility area, noise, frontier of war, unexplored areas, recent combats, and more [Rab02]. Influence fields have been used for several distinct types of games [DeL01], namely, in Ms. PacMan [WG08][SJ12], real-time strategy games [ML06][MQLL07], and turn-based games [LM05].

In spite of their versatility, influence fields suffer from a common problem known as local extrema. Assuming that a game agent moves in the direction of decreasing values of the field, a local extremum is refereed to as a local minimum, and when it moves in the direction of increasing values of the field it is refereed to as local maximum. A local extremum is a field's place where the field value of every single point in its neighborhood is either lower (local maximum) or higher (local minimum). Once at a local extremum an agent becomes trapped, being no longer able to leave [Goo00], as there is no place in its vicinity with either higher (when converging to higher field values) or lower (when converging to lower field values) field value to move to. In potential fields this occurs at a point where the total sum of vector forces reaches zero, while in influence maps this is the result of the non-existence of lower scalar values (around minimum) or higher scalar values (around maximum). In short, comparatively to pathfinders for motion planning, influence fields are not complete, as they do not ensure that a goal position is reached.

2.6 Field-Aware Pathfinders

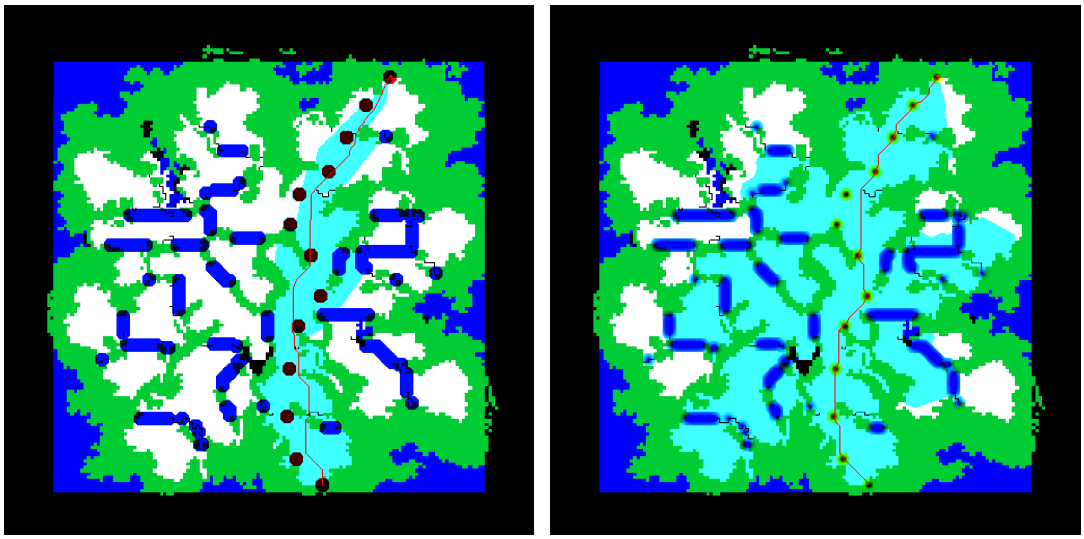
In regards to combining influence fields with pathfinders, let us mention a few works we found in the literature. Laue and Röfer [LR04] used a vector field for navigation of agents in a virtual world, resorting to a path search algorithm when in the vicinity of a local extremum in order to escape from it. Millington [Mil06] describes a formulation in which the influence value is directly added to the cost function.

2.6.1 Risk-Adverse Pathfinding

Paanakker [Paa08] further develops on Millington's idea with an implementation of modified Dijkstra's and A* algorithms, which consider a constant influence value in each propagator's influence circle. Specifically, Paanakker [Paa08] suggests using two weights α_1 and α_2 for a pathfinder cost function, as follows:

$$F(n) = \alpha_1 \,.\, (G(n) + H(n)) + \alpha_2 \,.\, I(n) \tag{2.7}$$

where $I(n)$ is the sum of influences of one or more influence maps. Paanakker calls to a pathfinder that satisfies Eq. (2.7) a risk-adverse pathfinder. However, our testing and assessment of this technique indicates that it does not ensure that a repeller is avoided or that an attractor is converged to. In fact, this technique works mostly for repellers only, requiring testing and tweaking per case. Fig. 2.6(a) illustrates a path found (in red) with a risk-adverse A* pathfinder, in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.



(a) (b)

Figure 2.6: Paths (in red) found in HOG2 crucible map of Warcraft 3 through: (a) risk-adverse A*; (b) constraint-aware navigation.

2.6.2 Constraint-Aware Pathfinding

Constraint-aware navigation was another attempt to integrate discrete pathfinders and influence maps, due to Kapadia et al. [Stu12, KNS+13, KK16], who modified the cost functions of Dijkstra's, A*, and AD* pathfinders in order to include constraints (repeller or attractor influence values). The cost function of this constraint-aware pathfinder is as follows:

$$F(n) = G(n-1) + M_c(n-1,n)\,.\,d(n-1,n) + H(n) \tag{2.8}$$

where $d(n-1,n)$ is the cost from $n-1$ to n, and $M_c(n-1,n)$ is the multiplier field given by the following expression:

$$M_c(n-1,n) \approx m_c\left(\frac{x_{n-1}+x_n}{2}\right) \tag{2.9}$$

where $m_c(x)$ is the aggregate cost multiplier that is expressed as follows:

$$m_c(x) = max\left(1, m_0 + \sum_{c \in C} m_c(x)\right) \tag{2.10}$$

where m_0 is a base multiplier with a value in the range $[3,10]$, which affects costs even in the absence of attractors/repellers (constraints $c \in C$), while x_n and $x-n-1$ stand for the locations of the current and previous nodes, respectively.

The constraints can be *near* (in the proximity) or *in* (inside) an attractor/repeller. For

near constraints $m_c(x)$ is given by the following function:

$$\begin{cases} m_c(x) = \overline{m}_c(x) & , |\overline{m}_c(x)| < \tau \\ 0 & , \text{otherwise} \end{cases} \tag{2.11}$$

where $\overline{m}_c(x)$ is given by:

$$\overline{m}_c(x) = -w(k1 + k2\,.\,r(c,x))^{-2} \tag{2.12}$$

where w is the constraint weight, $k_1 = 0.4$ and $k_2 = 0.5$, τ defines the influence radius of an attractor/repeller, $r(c,x)$ is the shortest position between the position x and the constraint c. For *in* constraints $m_c(x) = -w(k1)^{-2}$ for all positions within the constraint annotation and $m_c(x) = 0$ for all positions outside the annotation.

This, approach considers both attractors and repellers either for continuous or fixed value reppeling/attractive areas, like in the work of Paanakker. However, constraint-aware navigation is intended to converge to attractors and circumvent repellers in the path that would be found without influences. In order to do so, pathfinders resorting to this modification are forced to perform a larger search graph expansion. Fig. 2.6(b) shows a path found (in red) with a constraint-aware A* pathfinder, in the Warcraft 3 thecrucible map, while the expanded nodes are in cyan.

2.7 Further Remarks

Dijkstra's, A*, best-first search, and fringe search algorithms are not adaptive, though there a few variants of A* that address the problem of adaptivity in robotics, as it is the case of D*. Nevertheless, in video games, a pathfinder that addresses this issues is not a reality yet. Besides, there is no adaptive pathfinder within the family of general-purpose bounded suboptimal pathfinders, though these latter algorithms have been developed to restrain memory consumption by limiting the expansion of the search space, and simultaneously to speed up the performance in finding a path between two locations of a game map.

In turn, influence fields are a powerful technique for motion and tactical planning. However, they suffer from the problem of the local extremum, because they do not ensure path completeness, that is, they do not ensure that a path is found even if it exists.

Finally, a few influence-aware pathfinders have tried to combine pathfinding with influence fields to get a better control on the motion of NPCs and other game agents. However, they often fail to adapt to the placement of attractors and repellers. For example, risk-adverse pathfinders are not sensitive to the presence of attractors, while constraint-aware pathfinders are only sensitive to attractors and repellers if they are in the way generated by canonical pathfinders as A* or Dijkstra's.

As seen in the next chapters, we propose adaptive influence-aware pathfinders that truly integrate pathfinding with influence fields, that is, pathfinders that are sensitive to the dynamic placement of attractors and repellers over time, as a consequence of changes in the game world.

Chapter 3

Influence-based A* Algorithms

The family of A* pathfinding algorithms (i.e., Dijstra, A*, and best-first search) has evolved towards solving some of their drawbacks, namely: prompt adaptation to search space changes; reduction of the memory expenditure (or memory complexity); improvement of the time performance (or time complexity); and path smoothing to mimic the human-like movement behavior. This chapter describes a novel technique that, at least partially, solves those problems of A* pathfinders using influence fields. These influence-based A* algorithms are then compared to other pathfinders existing in the literature in order to better assess their performance.

3.1 Introduction

In games, path-finding has to do with finding a path between the current location node of an NPC and its goal location node. An NPC is an autonomous entity that is often considered as an enemy. Usually, an NPC is programmed in a loosely way to ensure a player has a chance to win a game or a game session. NPCs are not intelligent agents in literal terms, but they behave in a seamlessly plausible intelligent manner, particularly when they are chasing a player in the game world. For this plausible intelligent behavior much contributes motion planning algorithms for NPCs and agents [Nil98]. In fact, motion planning addresses the problem of decomposing the movement of these entities, from one place to another in the game world, into a series of smaller discrete motions. As described below, motion planning techniques divide into two major categories, namely: *path-finding* and *influence fields*.

3.1.1 Pathfinders

Path-finding operates on a search graph that describes the path network of the game world. The idea is to find a path (if it exists) between two given locations, preferably with the lowest cost; in other words, the pathfinder should be complete and optimal. Dijkstra, A*, and best-first search pathfinders [Dij59, HNR68, Poh70b] are three examples of complete pathfinders, but only the first two are optimal. These former pathfinders, also called canonical pathfinders, can be considered as belonging to the family of A* pathfinders. In fact, Dijkstra pathfinder is a particular A* pathfinder with the heuristic taking on the value 0, while the best-first search is a particular A* pathfinder with the cost-so-far taking on the value 0.

As seen in Chapter 1, the canonical A* pathfinders suffer from various limitations, among which we count on the lack of *path adaptivity*, *memory space and time consumption*, and lack of *path smoothness*. Some A* variants have been proposed to solve the *path adaptivity* problem, as it is the case of the lifelong planing A* (LPA*) [KLF04] and D* (also known as dynamic A*) family [Ste94, Ste95, KL05], i.e., whenever a node changes its position (e.g., a roundabout was moved to another site with more influx of traffic) or is removed (e.g., a bridge over a river blows up), or a new node is added to the graph (e.g., a street alleyway was constructed and flows into the middle of another street). However, these variants incur in additional memory and processing costs because at least the previous set of open nodes is taken into account to find the new path to the goal node.

Other A* variants are focused on solving the second problem above, i.e., reducing *memory occupancy* and improving *time performance*. These A* variants divide into two categories. The first reinforces constraints to the length of the open and closed sets used by A* [GMN94], and in some cases such sets are even discarded [Kor85, BEHS05]. The second focuses on altering the value of the cost function heuristic in order to constrain the graph expansion, hence resulting a reduction of memory and time, yet that at the cost of sacrificing the optimality of A*. This second category of A* variants is known as general-purpose bounded suboptimal searches, which include the static weighted A* [Poh70a], dynamic weighted A* [Poh73, TR09], AlphA* [Ree99], and optimistic and skeptical searches [TR08, TR10].

Finally, path smoothing is an attempt to correct the jagged motion generated by any path determined by a given pathfinder, i.e., the leading idea is to turn a piecewise linear path (i.e., consisting of a connected set of straight line segments) into a curved path [BMS04], as required to endow NPCs with an human-like movement, as well as to reproduce the motion of cars in racing games. The typical solution is to generate an approximating spline-based curved path for each piecewise linear path. But, using an spline that approximates the nodes of path may require further time-consuming operations to avoid collisions of an NPC with obstacles in the scene [Rab00]. Instead, our solution is using an interpolating cubic spline through the path nodes, so that obstacle avoidance need not be considered at all since we use a grid-based game world. As shown throughout this chapter, these three problems of canonical pathfinders can be solved at once by using influence fields. That is, instead of using specific solutions for particular problems, influence fields constitute a single solution for those problems jointly.

Note that, in games, it suffices to use complete pathfinders [BMS04]. Finding the shortest path is not a strict requirement in games, simply because such will turn into an advantage for NPCs over the player. That is, it is harder, not to say impossible, for a player to beat an NPC that acts optimally. Also, an optimal, complete pathfinder does not endow NPCs with the prone-to-error behavior of humans, which is in its essence complete, but not optimal. Therefore, it is acceptable to propose path-finding algorithms that sacrifice optimality for performance.

3.1.2 Influence fields

Unlike pathfinders, influence fields were not thought of to find a path between two locations, but solely to induce a steering motion on game entities that move around the environment. Influence fields were first employed in robotics [Kha85, Ark86, Ark87] to avoid collisions. Later on, they were used in games to achieve human-like movement of NPCs and game agents in real-time [Zob69, TBS04], and spatial and temporal reasoning aside from motion planning [SJ12, OSU+13, LCCFL13], having been used in several video games [DeL01], namely Ms. PacMan [WG08, SJ12], real-time strategy games [ML06, MQLL07], and turn-based games [LM05]. On the other hand, potential fields have been employed in decision-making in real-time strategy (RTS) games [HJ08a, HJ09b, HJ09a].

Recall that an *influence field* —also known in games as *influence map*— is essentially a function that has a single value (e.g., a weight or cost) for each point in game space and time, i.e., a concept known in mathematics as a scalar field [Apo69]. But, if the value is multivariate (or a vector), we end up having a vector field as in mathematics, which is known in games as *potential field* (or *flow field*). It happens that, as any other function, an influence field may possess one or more extrema (i.e., minima and/or maxima).

In *fields*, navigation is performed by traversing a grid of cells. Traversal is performed from each cell to its direct neighbor cell with either the largest or the lowest potential (vector) or influence (scalar) [HJ08b]. Depending on the employed convergence scheme —to the lowest (local minimum) or the highest (local maximum)— once at such a location there is no way to leave because there is no other place to converge, this is known as the local extremum problem. Consequently, *fields* do not ensure that a goal position is reached, when used for motion planing, if a local extremum is encountered and it is not the goal. In influence maps this is the result of the non-existence of lower weights (around minima) or higher weights (around maxima). For a pathfinder, this is equivalent to say that it is not complete.

3.1.3 Research Questions

Interestingly, some authors have used pathfinders to solve the problem of local minimum/maximum inherent to influence fields [LR04]. In this case, the pathfinder is only used to bring the game agent out the influence of a minimum/maximum. That is, path-finding is used to solve a problem of influence fields. In the present chapter, the approach is the opposite, i.e., how influence fields can be used to solve the problems of path-finding.

Therefore, the main research question (RQ) underlying our research work is, as follows:

Is it feasible to couple influence fields together with distance-based cost functions to solve the problems of canonical pathfinders?

3.1.4 Contributions

In short, the problem of motion planning in games can be addressed using either *path-finding* or *fields* (i.e., potential fields or influence maps). The main contribution of this chapter is a novel technique that allows to combine canonical pathfinders (i.e., Dijkstra's, A*, and best-first search) with continuous influence maps. More specifically, the main contribution lies in the fusion of these motion planning techniques into a new class of pathfinders, hereafter called *influence-based adaptive pathfinders*, i.e., pathfinders that adapt to environment changes in the game; these changes translate themselves into altering the influence map and consequently costs/priorities of graph nodes without changing the search graph directly (adding or removing nodes).

As a first consequence, the annoying problem of local extrema in *fields* is eliminated definitely. As a second consequence, the paths produced by pathfinders can be smoothed during search, thus, looking much more human-like in the sense that NPCs avoid obstacles in the game space. Finally, there may be a noticeable improvement in performance in terms of memory space and time consumption in comparison to canonical pathfinders, i.e., depending on the field one may need to traverse a smaller number of graph nodes, thus reducing computation and memory storage and access, while the convergence is faster towards the solution.

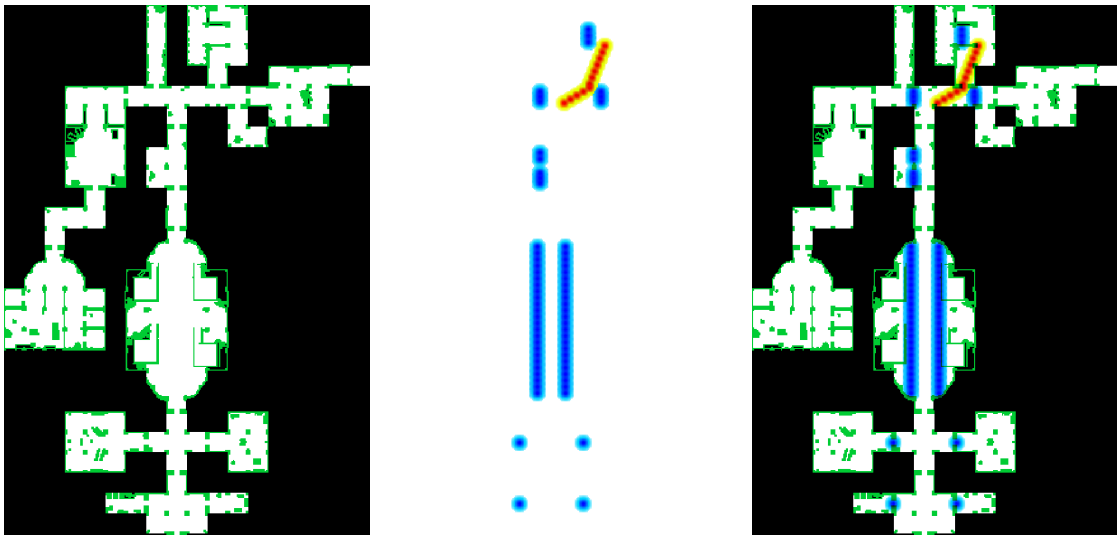
3.2 Theory of Fields

Potential fields work by strategically placing attractive and repulsive force propagators in the game world. A propagator is the center of a field, that delivers a potential field, that gradually fades to zero [HJ08b]. In general, if a field charge is attractive, objects move towards it; but if its charge is repulsive, objects move away from it. Therefore, a potential field is an emulated electromagnetic-like field that results from summing up subsidiary potential fields generated by a number of attractive and repulsive propagators in the game world. These propagators may be tied to static and dynamic game entities (e.g., a skyscraper as a static entity and a bot as a dynamic entity that moves around) or not.

The term 'influence map' in games is known as 'scalar field' in mathematics. Scalar influence maps are a scalar version of potential fields, where vectors are replaced by a weight or cost. Influence maps are much more generic than potential fields, as they can be used for spatial and temporal reasoning aside from motion planning [SJ12, OSU+13, LCCFL13]. Spatial and temporal reasoning has to do with the analysis of terrain, as well as understanding events that occur over time.

As explained further ahead, we use attractors (or sinks) and repellers (or sources) to guide an NPC in its way to the goal, avoiding obstacles at the same time. Recall that a discrete motion planner is a pathfinder plus an obstacle avoidance technique. The

obstacle avoidance technique used in this chapter is the influence map, which is a computational representation of a scalar field. An attractor is a local minimum of a scalar field, while a repeller is a local maximum of a scalar field. Attractors and repellers are here called propagators. In Fig. 3.1, we have 12 attractors in red and 66 repellers in blue. Intuitively, a *scalar field* ties a scalar value to every point in a space (e.g., 3D Euclidean space or $\mathbb{R}^3$). Even taking into account that the game world $D \subset \mathbb{R}^3$ is bounded in size, we know that the number of points of D is uncountable, so we need to discretize the D into a number of cubes, so that we then calculate the value of the scalar field at each corner of every single cube. For the sake of convenience, we consider that D represents the terrain of the game world, so that it is tiled in terms of squares, not in terms of cubes.



(a) (b) (c)

Figure 3.1: Representation of a grid-based game world: (a) game map; (b) influence map with propagators; (c) game map together with influence map.

A *scalar field* in $\mathbb{R}^2$ is generated by a real bivariate function $f : \mathbb{R}^2 \to \mathbb{R}$, that is, f is defined at every single point of $\mathbb{R}^2$. In this chapter, we use a Gaussian function f_i to model the scalar field generated by each propagator i, which is given by:

$$f_i(\mathbf{p}) = a\, e^{-d_i^2 \cdot \delta_i^2} \tag{3.1}$$

where a stands for the amplitude (i.e., height) of the Gaussian, d_i the distance of an arbitrary point $\mathbf{p} \in \mathbb{R}^2$ to the location $\mathbf{p}_i$ of the propagator i, and δ_i the decay factor of the Gaussian with the distance in relation to the location of the propagator i. More specifically, we have:

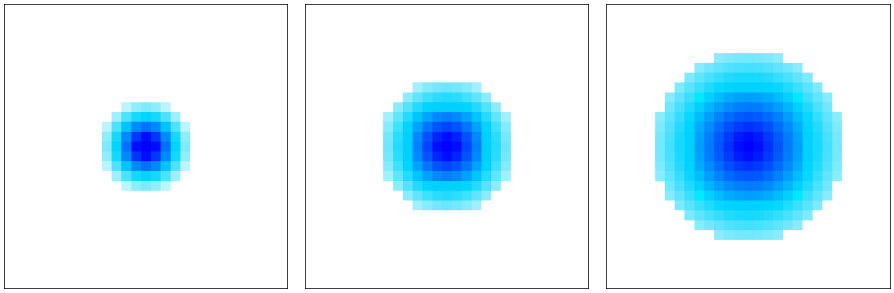
$$a = \frac{1}{2\pi\sigma^2}, \qquad d_i = \|\mathbf{p} - \mathbf{p}_i\|, \qquad \delta_i = \frac{1}{\sqrt{2}\,\sigma} \tag{3.2}$$

where σ denotes the standard deviation. Fig. 3.2 shows us the effect of the decay δ_i on the influence area of a propagator so that, the bigger is the decay, the lesser the influence area of a propagator.

Note that each function f_i represents the decaying behavior of the scalar field of the propagator i with the distance. That is, the propagator is stronger at its location than at any other point in the game world. Note that the value of f_i for an attractor is always negative, while it is positive for a repeller, that is, we have to invert the sign of f_i for an attractor. Summing up the scalar fields of all propagators results in a scalar field F of the game world, as follows:

$$I(\mathbf{p}) = \sum_{i=0}^{n-1} f_i(\mathbf{p}) \tag{3.3}$$

where n is the total number of propagators existing in the game world.



(a) (b) (c)

Figure 3.2: Different values for the decay δ of a repeller: (a) with $\sigma = \sqrt{5}$; (b) with $\sigma = \sqrt{10}$; (c) with $\sigma = \sqrt{20}$.

The main problem with any Gaussian propagator f_i is that its kernel is unbounded, i.e., it contributes to the value of I in Eq. (3.3) at every point of the game world. As a consequence, when a propagator moves, the overall scalar field I has to be recalculated for every corner of the terrain tiles. To overcome this problem, one must use truncated Gaussian propagators. For every truncated Gaussian propagator, it must be considered a small threshold τ (e.g., $\tau = 0.1$) below which the value of f_i is always zero. Doing so, it is straightforward to determine the influence radius of each propagator from Eq. (3.1), as follows:

$$\tau = a\, e^{-d_i^2 \cdot \delta_i^2} \tag{3.4}$$

and, by manipulating Eq. (3.4), we get the influence radius of the propagator i, which is given by:

$$d_i = \frac{1}{\delta}\sqrt{\ln(\frac{a}{\tau})} \tag{3.5}$$

So, given the tile size Δ, we can say that the square influence neighborhood of each propagator i is a $2\lceil\frac{d_i}{\Delta}\rceil \times 2\lceil\frac{d_i}{\Delta}\rceil$ neighborhood centered at $\mathbf{p}_i$, where $\lceil\frac{d_i}{\Delta}\rceil$ is smallest integer not less than $\frac{d_i}{\Delta}$. This means that the values of the influence neighborhood tiles may remain unchanged from the time they are calculated through Eq. (3.1), regardless of whether the propagator moves itself in the game world or not.

When a propagator moves around in the game world, what changes is the *influence map* of the game, which is a discrete representation of the overall scalar field I given by Eq. (3.3). We say 'discrete' because, after partitioning the game terrain into square tiles, I is evaluated at the center of each tile. Note that the changes in the influence map are local because they are confined to tiles under the influence of a given propagator.

3.3 Influence-based Pathfinders

While an influence field over a game world is important as an obstacle avoidance technique for NPCs, the core of any discrete motion planner is a pathfinder. Pathfinders address the problem of searching a path in a graph. In games, path-finding is used to find routes between two separate nodes of a graph, and to help in the formulation of strategies for artificial intelligence (AI) agents (e.g., chess games resort to path-finding to formulate strategies).

However, not finding the best path is not problematic, as the best path is not a strict requirement in games [BMS04], because an optimal and complete pathfinder does not simulate the prone-to-error human nature. In fact, humans perform complete but not necessarily optimal path-finding, i.e., an optimal and complete pathfinder will be perceived by players as an unbeatable, or hard to beat, opponent, not to say super-human. In other words, a pathfinder providing sub-optimal solutions is a requirement to control the degree of difficulty across successive game levels, making it possible for the players to sometimes beat the AI, or at least have a fighting chance. In the next two subsections, we describe two complete pathfinders capable of avoiding obstacles, as humans do in practice.

3.3.1 Influence-based Dijkstra Pathfinder

Dijkstra's algorithm leading idea is the minimum cost to get from the start node to the currently being evaluated node, i.e., the value $G(n)$ of the cost-so-far (CSF); hence the objective function is then given by:

$$F(n) = G(n) \tag{3.6}$$

The cost-so-far value $G(n)$ is a single metric that features the lowest cost to get to each node in the graph. Therefore, for each explored/visited node, we have to save its current shortest path to the start node. In fact, the cost to get to a node may be altered if a lower cost path is found; in these circumstances, we are assuming that nodes may be revisited/reevaluated. The algorithm only stops, when all possible minimum-cost paths (one per node) have been evaluated. This is costly in terms of memory and time consumption because all nodes are evaluated often more than once; hence, Dijkstra's

algorithm is both complete and optimal.

In a game world tiled into squares, where a square is a node connected to its eight neighbor nodes (squares), the required memory and time resources to deal with the needs of Dijkstra's algorithm become unfordable for the real-time requirements. A way of mitigating the consumption of resources by Dijkstra's algorithm in games is to use a point-to-point search version of Dijkstra's. i.e., the algorithm stops when one of the explored nodes is the goal node. However, this Dijkstra's variant will not ensure that path found to the goal is optimal.

The present chapter introduces another way of reducing the resources allocated to Dijkstra's algorithm. This novel Dijkstra's variant is based on the concept of influence map, as described in the sequel. The leading idea of combining a point-to-point search version of Dijkstra's algorithm and an influence map translates into an augmented cost function, as follows:

$$F(n) = \begin{cases} \left| \dfrac{I_{MIN} - I(n)}{I_{MIN}} \right| \cdot d & , I(n) < -\tau \\ G(n) & , I(n) \in]-\tau, \tau[\\ G(n) + I(n) \cdot N \cdot d & , I(n) > \tau \end{cases} \tag{3.7}$$

with $F(n) \geq 0$, $\forall n \in \mathbb{N}$, and where $I(n)$ is the influence value at the currently being evaluated node as yield by Eq. (3.3), I_{MIN} in the most negative influence value in the map, which corresponds to the value of the influence at the center of some a attractor (even when two or more attractors have the same location) in the influence map, d is the maximum Euclidean distance between two connected neighboring nodes (i.e., in the case of a regular grid spanning the game space, this corresponds to the length of the diagonal connecting two grid nodes), and N in the total of nodes in the graph.

The restriction $F(n) \geq 0$, $\forall n \in \mathbb{N}$, is a requirement for Dijkstra, as most pathfinders, because it is incapable of dealing with both positive and negative weighted graphs [Dij59]. Eq. (3.7) topmost branch deals with nodes under the influence of an attractor (i.e. attracting nodes), at center of which the function I attains a local negative minimum. Accordingly, $F(n)$ vanishes at the center of any attractor. Eq. (3.7) bottommost branch concerns nodes under the influence of a repeller; the function I takes on a local positive maximum value at the center of such a repeller. Note that, Eq. (3.7) bottommost branch was designed in order to endow each repelling node (i.e., a node under the influence of a repeller) with less priority relative to any other type of node in the set of open nodes. Finally, Eq. (3.7) middle branch applies to any other node, as usual with Dijkstra's.

The cost function given by Eq. (3.7) is subtle in the sense that it changes the typical behavior of Dijkstra's pathfinder (point-to-point variant), as shown in Fig. 3.3. In Figs. 3.3(a) and (e), the game world has no propagators nor obstacles, so that the path in dark slate gray tends to be linear from the start node to the goal node. In Figs. 3.3(b) and (f), the game world has two attractors, with the path passing through them. Note

that the path does not stop at any attractor. This means that attractors no longer work as local minimum traps for influence maps, and this is so because, in conformity with the expansive search nature of Dijkstra's, the travelling agent keeps walking toward the goal node. In Figs. 3.3(c) and (g), the game world has two repellers, so that the agent avoids them in its way toward the goal node. This means that repellers can be tied to eventual obstacles in the scene, enabling the traveler to move in the game world without bumping into them. Figs. 3.3(d) and (h) show the behavior of the pathfinder in the presence of attractors and repellers. In Figs. 3.3(a) to (h) blue lines illustrate connections among nodes whose neighbors were evaluated, regardless if they where or not placed into the open set.

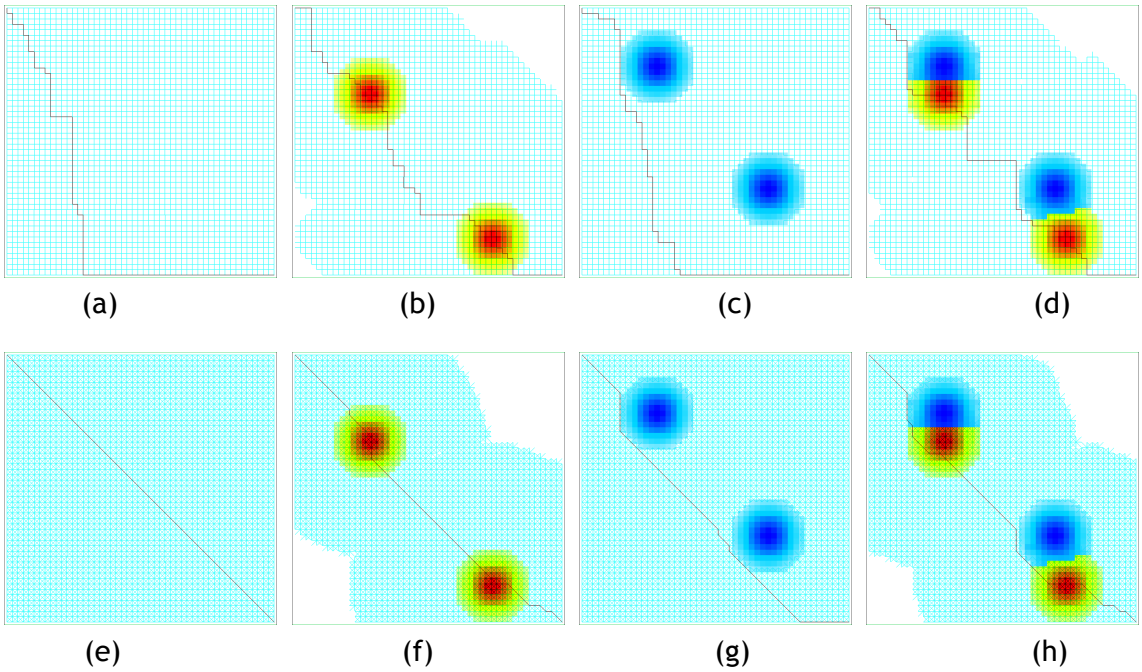


Figure 3.3: Dijkstra algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

3.3.2 Influence-based A* Pathfinder

The leading idea of the A* algorithm can be seen as an extension to Dijkstra's algorithm in the sense that the objective function comprises not only the cost-so-far function $G(n)$, but also an heuristic function $H(n)$, as follows:

$$F(n) = G(n) + H(n) \tag{3.8}$$

The heuristic $H(n)$ represents the likelihood of the current node converging faster to the goal, which is based on an estimate of the cost to the goal. For example, a possible

heuristic is the Euclidean distance from the current node to goal node. In other words, $G(n)$ refers to the cost of the current node to the start node, while $H(n)$ denotes the cost of the current node to the goal node.

The use of an heuristic means that A* is solely optimal if the heuristic is appropriate, i.e., the heuristic value must always be bellow the real cost from a node to the goal. The heuristic serves two purposes. Firstly, it serves to guiding the search flow in order to speed up convergence toward the goal node in relation to Dijkstra's algorithm. Secondly, it serves to reducing the amount of traversed nodes in the way toward the goal node when compared to Dijkstra's algorithm. Roughly, A* evaluates a number of nodes that is less than 50% of those evaluated by Dijkstra's algorithm, yet at the cost of an increasing in computation resulting from the further evaluation of $H(n)$. That is, the cost of visiting less number of nodes is not a synonym of a faster pathfinder.

In respect to the merging of A* with the influence map generated by the Gaussian function I, we end up having the following cost function:

$$F(n) = \begin{cases} \left|\dfrac{I_{MIN} - I(n)}{I_{MIN}}\right| \cdot d & , I(n) < -\tau \\ G(n) + H(n) & , I(n) \in]-\tau, \tau[\\ G(n) + I(n) \cdot N \cdot d & , I(n) > \tau \end{cases} \tag{3.9}$$

which is identical to $F(n)$ (cf. Eq. (3.7) topmost branch), with $G(n)$ in Eq. (3.7) middle branch replaced by $G(n)+H(n)$ in Eq. (3.9) middle branch. Recall that, similar to Dijkstra's, A* is also incapable of dealing with both positive and negative weighted graphs. Moreover, the value of I at each node needs not to be evaluated at runtime, because it is supposed to be set for the entire influence map before running the pathfinder, unless propagators are added or removed from the influence map or any propagator is moving around in the game world. As shown further ahead, the impact of I on the pathfinder performance is more noticeable in A* than Dijkstra's, i.e., memory space and time consumption.

It is worthy noting that it was assumed that attractive influence has negative values, so the pathfinder converges to nodes with more negative influence values. That is to say that, attractive influence will decrease the cost to travel among nodes, and repulsive influence will raise it. Consequently, the nodes influenced by attractive influence will be examined first, and nodes with repulsive influence will be examined later or not at all. Figs. 3.4(a) to (h) illustrate the behavior of the pathfinder either in the presence or absence of propagators.

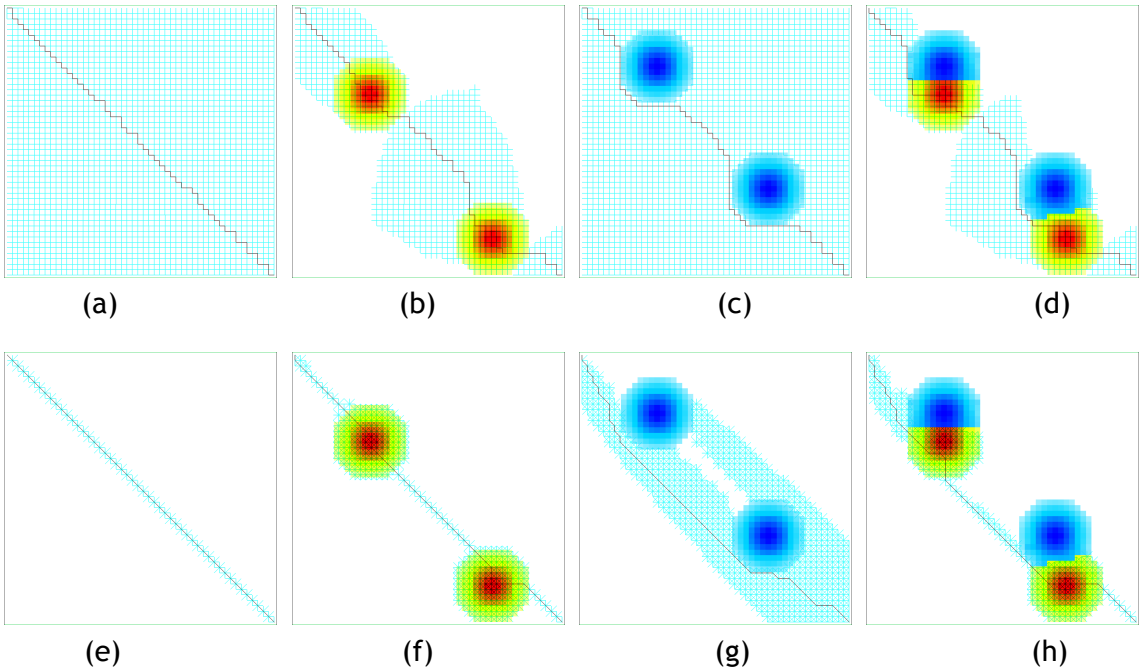


Figure 3.4: A* algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

3.3.3 Influence-based Best-First Search Pathfinder

The leading idea of the best-first search algorithm can be seen as A* without the cost-so-far function $G(n)$:

$$F(n) = H(n) \tag{3.10}$$

The removal of the cost-so-far function means that best-first search performs an heuristics-based depth-first search. This, results in an less memory consuming and often faster pathfinder, than Dijkstra and A*. However, it also means best-first search cannot ensure the found path is optimal.

In regards to the merging of best-first search with the influence map generated by the Gaussian function I, we end up having the following cost function:

$$F(n) = \begin{cases} \left| \dfrac{I_{MIN} - I(n)}{I_{MIN}} \right| \cdot d & , I(n) < -\tau \\ H(n) & , I(n) \in]-\tau, \tau[\\ I(n) \cdot N \cdot d & , I(n) > \tau \end{cases} \tag{3.11}$$

whose topmost branch is identical to the one given by Eq. (3.9), middle branch is also identical to the one given by Eq. (3.9) but with $G(n) = 0$, and bottom branch is given

by the one of Eq. (3.9) but with $G(n) = 0$.

Recall that, similar to A*, best-first search is also incapable of dealing with both positive and negative weighted graphs. Moreover, the value of I at each node needs not to be evaluated at runtime, because it is supposed to be set for the entire influence map before running the pathfinder, unless propagators are added or removed from the influence map or any propagator is moving around in the game world. As shown further ahead, the impact of I on the pathfinder performance is more noticeable in best-first search than A*, i.e., memory space and time consumption. Figs. 3.5(a) to (h) illustrate the behavior of the pathfinder either in the presence or absence of propagators.

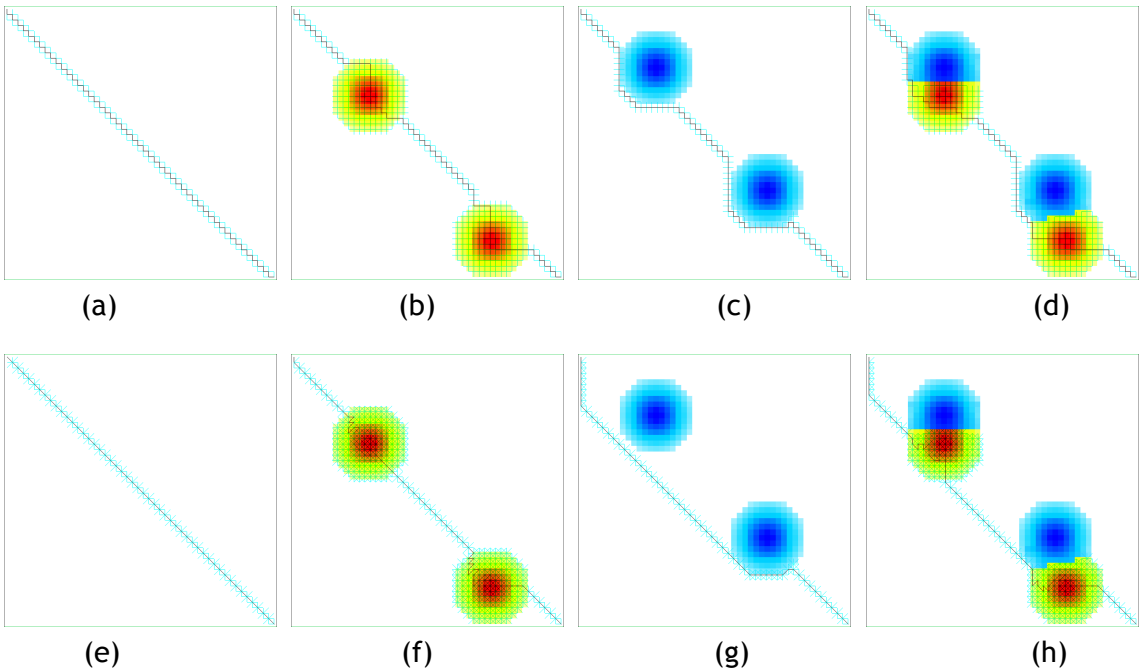


Figure 3.5: Best-first search algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

3.4 Experimental Results

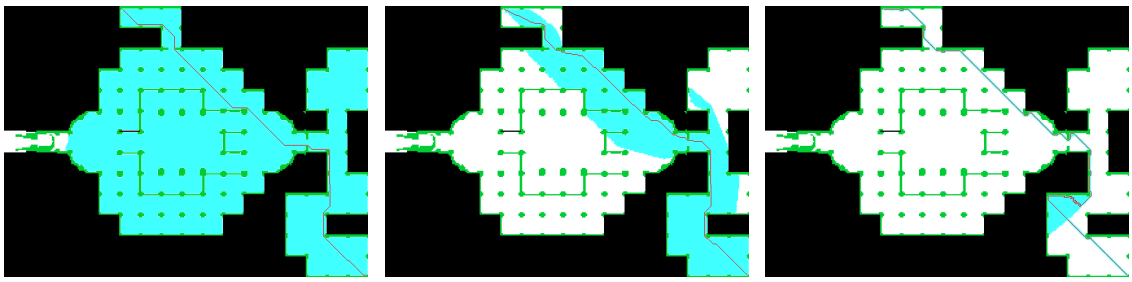
This section carries out a comparative analysis of Dijkstra (point-to-point variant), A* and best-first search (`BestFS`) pathfinding algorithms in relation to their influence-based variants, here called Dijkstra+, A*+, and BestFS+. Also, a comparative analysis is made against five general purpose suboptimal bounded search algorithms, namely, static weighted A* (`swA*`), revised dynamic weighted A* (`rdwA*`), AlphA*, and Optimistic and Skeptical searches (`OS` and `SS`), using a bound value of 1.5 as in [TR10].

3.4.1 Software/Hardware Setup

The nine pathfinders were implemented in the Java programming language. The tests were performed on a desktop computer running a Windows 7 64-bit Professional operating system, equipped with an Intel Core i7 920 @ 2.67GHz processor processor, 8 GB Triple-Channel DDR3 de RAM, and a NVIDIA GeForce GTX 295 graphics card with 896MB GDDR3 RAM.

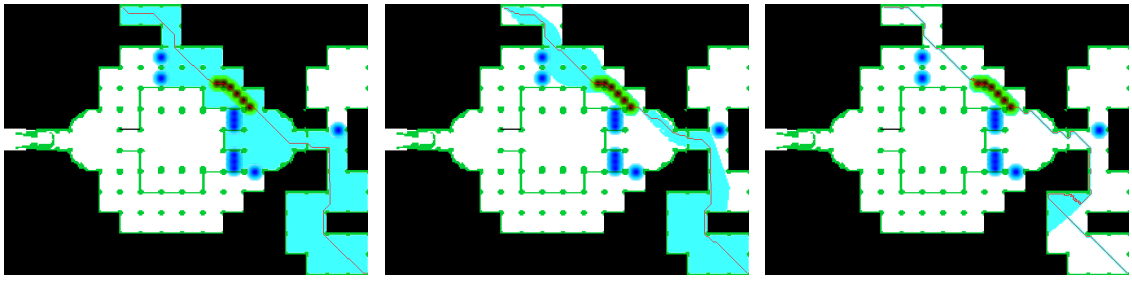
3.4.2 HOG Map Dataset

For testing, we used a dataset of 20 game maps taken from the HOG2 map repository (`http://movingai.com/benchmarks`), 10 out of which belong to *Dragon Age: Origins* (`http://www.dragonage.com/`), while the remaining 10 maps concern *Warcraft 3* (`http://us.blizzard.com/en-us/games/war3/`). An example map without influence of these games is shown in Fig. 3.6, where cyan nodes denote the search space; the same map with attractors and repellers (i.e., with influence) is depicted in Fig. 3.7. Recall that *Dragon Age: Origins* is a role playing game (RPG), which mostly consists of indoor dungeon-like scenarios. In turn, *Warcraft 3* is a real-time strategy (RTS) game, which essentially is an outdoor game with open scenarios, mostly swamps or islands. The HOG repository does not contain any dataset for first-person shooter (FPS) games.



(a) Dijkstra. (b) A*. (c) Best-first search.

Figure 3.6: Identical paths found by Dijkstra's, A*, and best-first search algorithms (without influence of attractors and repellers) within the *Arena2* map of Dragon Age Origins.



(a) Dijkstra. (b) A*. (c) Best-first search.

Figure 3.7: Distinct paths found by Dijkstra's, A*, and best-first search algorithms (with influence of attractors in red and repellers in blue) within the *Arena2* map of Dragon Age Origins.

3.4.3 Grid-Based Game Map

Dragon Age: Origins and *Warcraft 3* use grid-based maps, i.e., each map is a quadrangle divided into square tiles, sometimes called cells. Each square cell is surrounded by 8 square cells, except if it is a boundary cell of the map; a corner cell has 3 neighbor cells, while an edge cell has 5 neighbor cells. In terms of programming, a map is encoded as a 2-dimensional array of size $l \times w$, where l stands for the number of cells along the length, while w is the number of cells along the width of the map.

Besides, not all cells are passable; for example, the black cells in *Arena2* map shown in Fig. 3.6 are not passable because they are out-of-bounds cells. Dark blue and green cells correspond to impassable cells either. In indoor scenarios of *Dragon Age: Origins*, such green cells correspond to walls and plants, as well as other decorative elements; in outdoor scenarios as those of *Warcraft 3*, green nodes denote forests and other obstacles, while blue cells correspond to deep water. The white cells are known as passable cells in indoor and outdoor scenarios. Nevertheless, light blue cells are also passable, though with a higher cost because they denote shallow water of lakes, rivers, and oceans.

3.4.4 Grid-Based Influence Map

We have generated an influence map for each game map with the same size, which is also encoded as a 2-dimensional array of size $l \times w$. Attractors (red, yellow and, orange tones) and repellers (blue tones) were then placed within the map, as illustrated in Fig. 3.7 (see also Fig. 3.1). The placement of propagators was made in an automated manner, in order to guarantee that the paths produced by pathfinders with and without influence would be distinct (cf. Fig. 3.7), but not far away from each other.

Specifically, repellers were placed to avoid regions where the search would expand when using the canonical Dijkstra algorithm without influences, while attractors were placed in order to force field of view traversals of the encountered path when using the canonical Dijkstra algorithm. For repellers this was done to illustrate cases where during game an area may be blocked or is to be avoided prior to path search due to several reasons, e.g., concentration of enemy units. Attractors where placed to illustrate how instead of using field of view graphs one may use attractors instead.

The number of attractors vary from a map to another in the range $[5, 52]$; more specifically, we used 5 attractors in the map *brc202d* (belonging to *Dragon Ages: Origins*), and 52 attractors in the map *gardenofwar* (belonging to *Warcraft 3*). In turn, the number of repellers vary in the range $[8, 124]$; in particular, we used 8 repellers in the map *brc204d* (belonging to *Dragon Ages: Origins*), and 124 in the map *isleofdread* (belonging to *Warcraft 3*). For simplicity, tests were performed using attractors defined by the parameters $\sigma = -\sqrt{10}$ and $\tau = 0.1$, while repellers were parameterized through $\sigma = \sqrt{10}$ and $\tau = 0.1$.

3.4.5 Graph Representation

We have also generated a non-directed graph for each game map. The nodes of map graph represent passable cells of the map, while the edges connect neighbor passable cells. A graph density is given by percentage of passable cells, as follows:

$$p = \frac{k}{l \,.\, w} \tag{3.12}$$

where k stands for the number of passable cells, while $l \,.\, w$ denotes the total number of cells of the map. More specifically, the value of p ranges in the interval $[12.13\%, 48.23\%]$ in the case of *Dragon Age: Origins*; in regards to *Warcraft 3*, we have $p \in [29.7\%, 61.4\%]$. That is, generally speaking, the outdoor maps of *Warcraft 3* have more passable cells than the indoor maps of *Dragon Age: Origins*.

3.4.6 Graph Search-Based Paths

In the chapter, we assume that searching a graph from a start node to a goal node is carried out using diagonals. This means that at the time of evaluating the cost function for a given cell node, we consider not only the four neighboring cell nodes aligned with the x and y axes, but also those four neighboring cell nodes aligned with its diagonals, i.e., we consider all of its 8 neighboring cell nodes. We have followed this *8-neighbor search strategy* because the path between a start node and a goal node is shorter using diagonals than not using diagonals.

Besides, we performed for each map three path searches always in the same order (`Test 1`). First, the *potentially longest paths* for each game map, i.e., the start and goal positions were placed on opposite locations of each map. More specifically, we always used the left potentially longest (LPL) path for each map, which is defined by the right bottommost node as the start node and the left topmost node as the goal node. Second, a path between the closest nodes to each of the positions of two of randomly selected attractor centers. Third, a path between the closest nodes to each of the positions two randomly selected attractor centers distinct from the ones used in the second search. The results presented in Figs.3.12 to 3.16 are the mean values of these three tests, for the expanded/evaluated nodes, memory consumption, and total time for either pathfinder. This was made to compare the performance for distinct path searches per map of using general purpose bounded searches vs canonical pathfinders vs influence map-based pathfinders. However, in order to reuse the placed propagators the path start and goal nodes had to be chosen from attractors as refereed. Alternatively, propagators could be placed randomly but there would be no assurance they would be relevant (traversed) during path search.

Additionally, a distinct path search was made (`Test 2`). In this case it was also per-

formed a search for the LPL path. The distinction is that first the search is performed using influence-based pathfinders and during search we remove one of the attractors randomly. Secondly, we compare this test with performing two searches (summing the values memory space, time consumption, and nodes expanded) using either canonical Dijkstra, A*, and BestFS or the five general purpose bounded searches. In the second search however the closest node to the center of the propagator removed in the first test is removed from the search graph. The intent of this test is to compare the removal of a node with the removal of a propagator, in order to better illustrate what differs among the proposed technique and some prior pathfinder purely graph centric approaches.

As shown in Fig. 3.6, even dough the paths are both optimal, the nodes belonging to each LPL path is not identical for Dijkstra and A* pathfinders. In addition, the paths generated by Dijkstra+, A*+, and BestFS+ variants produce similar, but not identical, paths, as illustrated in Fig. 3.7. In general, Dijkstra+, A*+, and BestFS+ pathfinders produce longer paths than Dijkstra and A* because the latter ones determine (tend to determine) the shortest path between two nodes, and the attractors and repellers change the course of such a path. However, note that the expansion of the graph is altered significantly in Dijkstra and A*. It is clear that this seams to imply that the memory space occupancy and time performance of Dijkstra, A*, and BestFS may be higher in their canonical versions than in their influence-based variants.

In Figs. 3.8 and 3.9 the number of expanded/evaluated nodes are presented for each map for either pathfinder used in `Test 1`, while for `Test 2` these results are illustrated in Figs. 3.10 and 3.11.

By observing Figs. 3.8, 3.9, 3.10, and 3.11, several conclusions can be made. First, there is a clear reduction in evaluated nodes when comparing canonical pathfinders with their influence-based versions. Second, a non specific general purpose bounded search is always in either Dragon Age Origins or Warcraft 3 maps the pathfinder with the lowest expanded nodes. However, this gap was as expected much lower in `Test 2` than in the `Test 1`. Also, the Warcraft 3 maps topology or navigability from one node to either node in the search graph appears to make this gap smaller in either test cases.

3.4.7 Memory Space Occupancy

The memory space occupancy of a pathfinder essentially depends on the number of nodes that have passed by the open set, some of which are now closed nodes. Thus, the total memory consumption comprises the memory occupied by the nodes that passed by the open set plus the memory needed to hold the set of path nodes; the closed nodes have already been open nodes, so that we do not need to account for them. Recall that each iteration of a pathfinder evaluates the cost function of a single node retrieved from the open set, after which the node is moved into the closed set. In other words, each pathfinder's iteration evaluates a closed node, being its neighbors put into the open

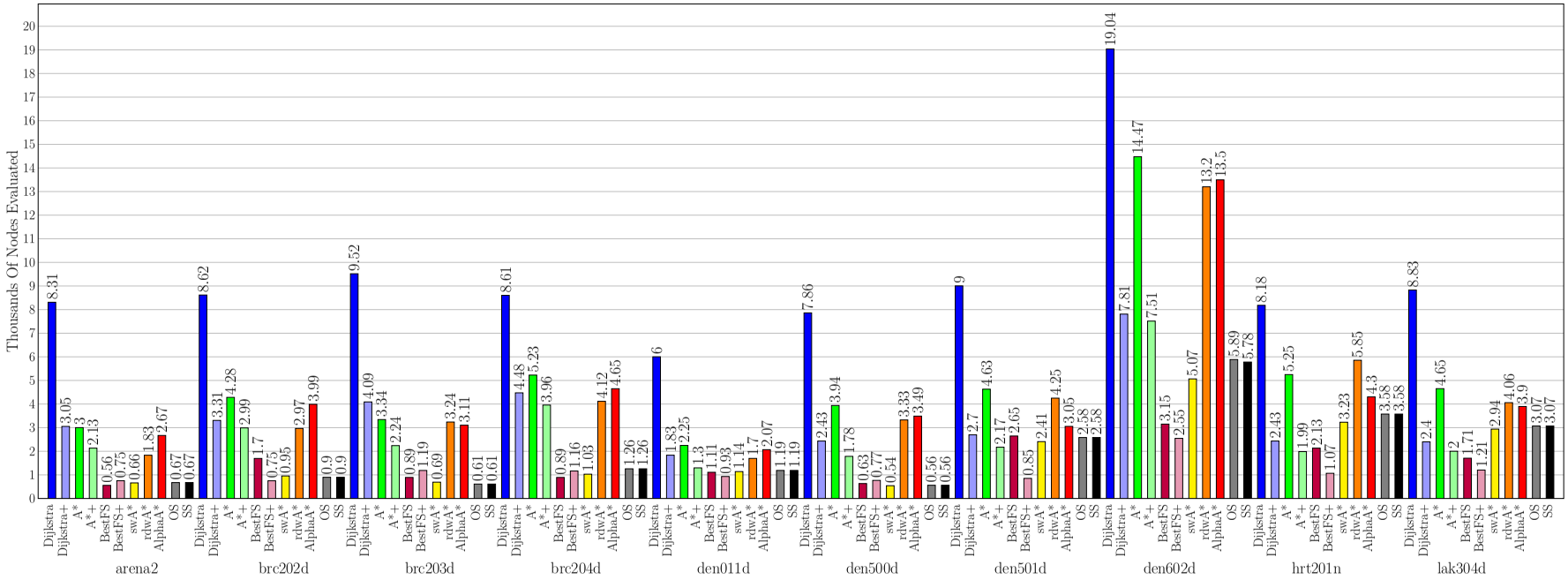


Figure 3.8: *Dragon Age: Origins* number of evaluated nodes (test 1).

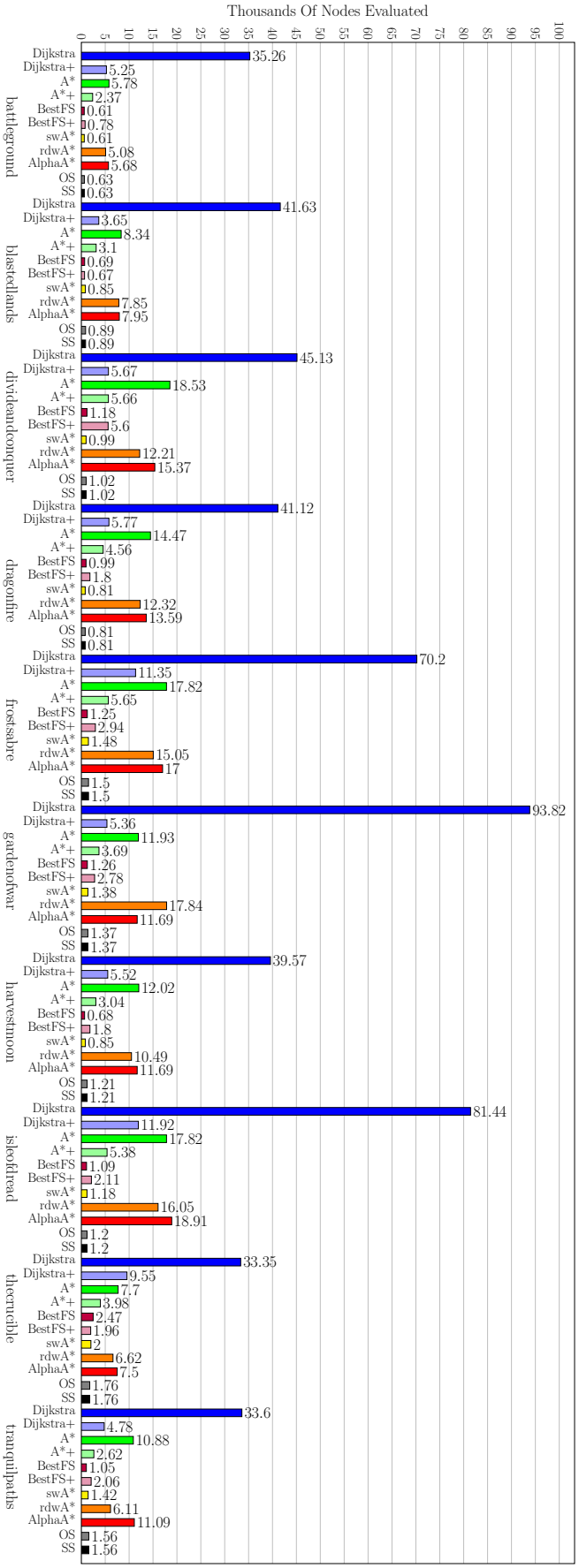


Figure 3.9: *Warcraft 3*'s number of evaluated nodes (test 1).

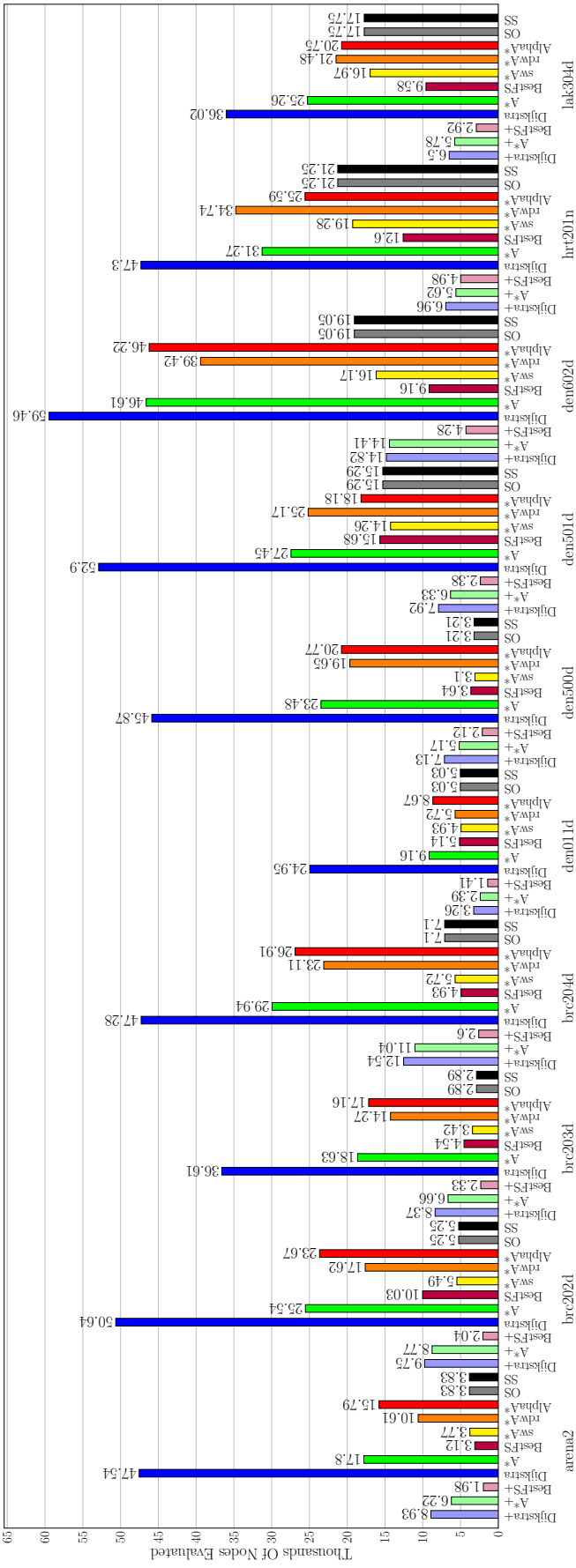


Figure 3.10: *Dragon Age: Origins* number of evaluated nodes (test 2).

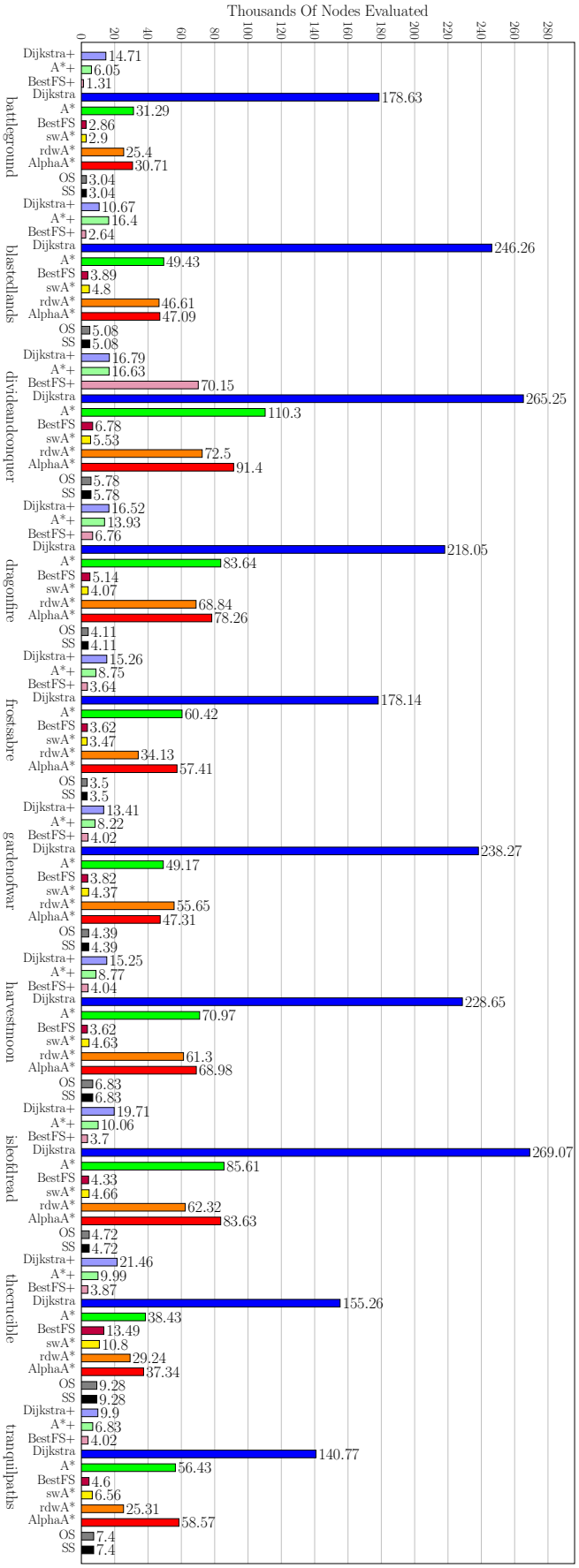


Figure 3.11: *Warcraft 3*'s number of evaluated nodes (test 2).

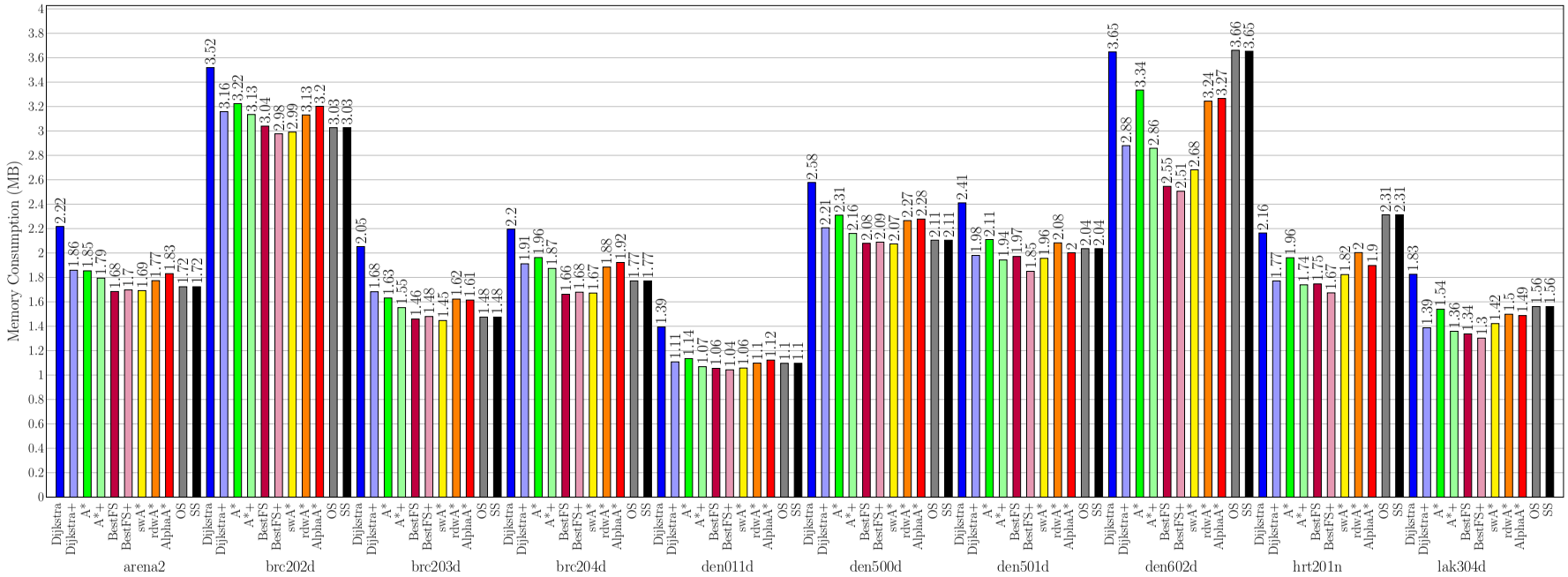


Figure 3.12: *Dragon Age: Origins* memory consumption results (test 1).

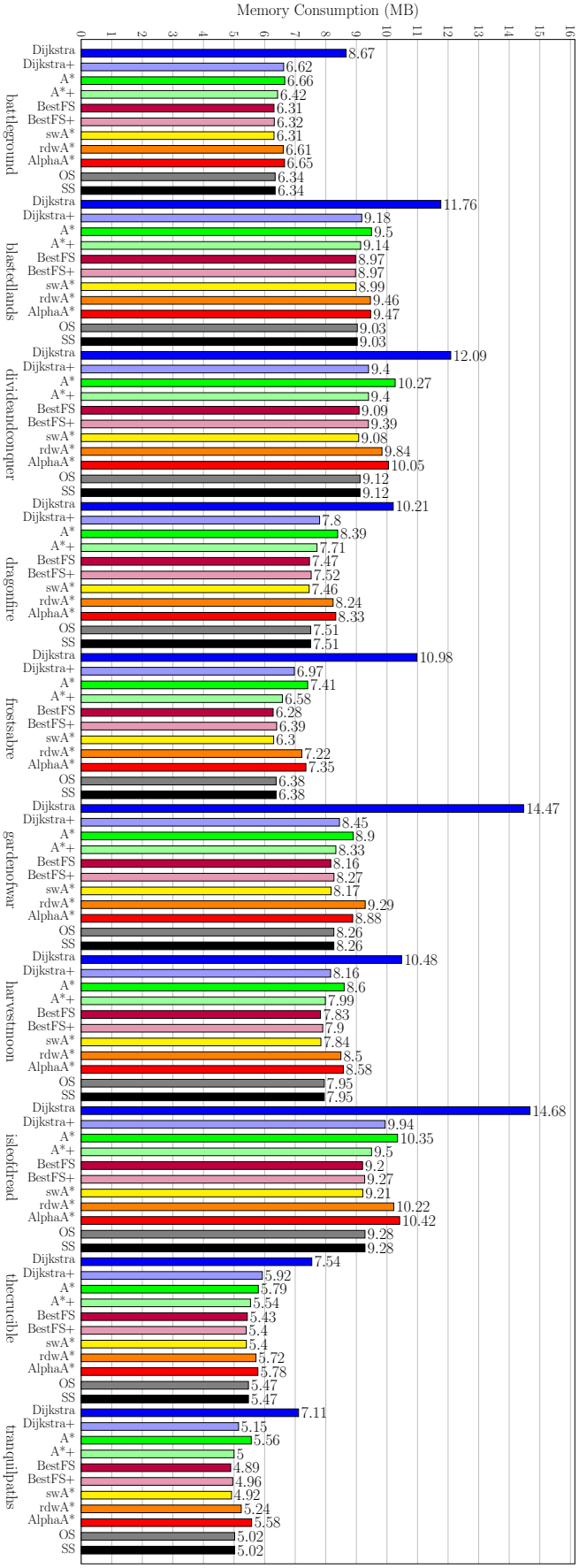


Figure 3.13: *Warcraft 3*'s memory consumption results (test 1).

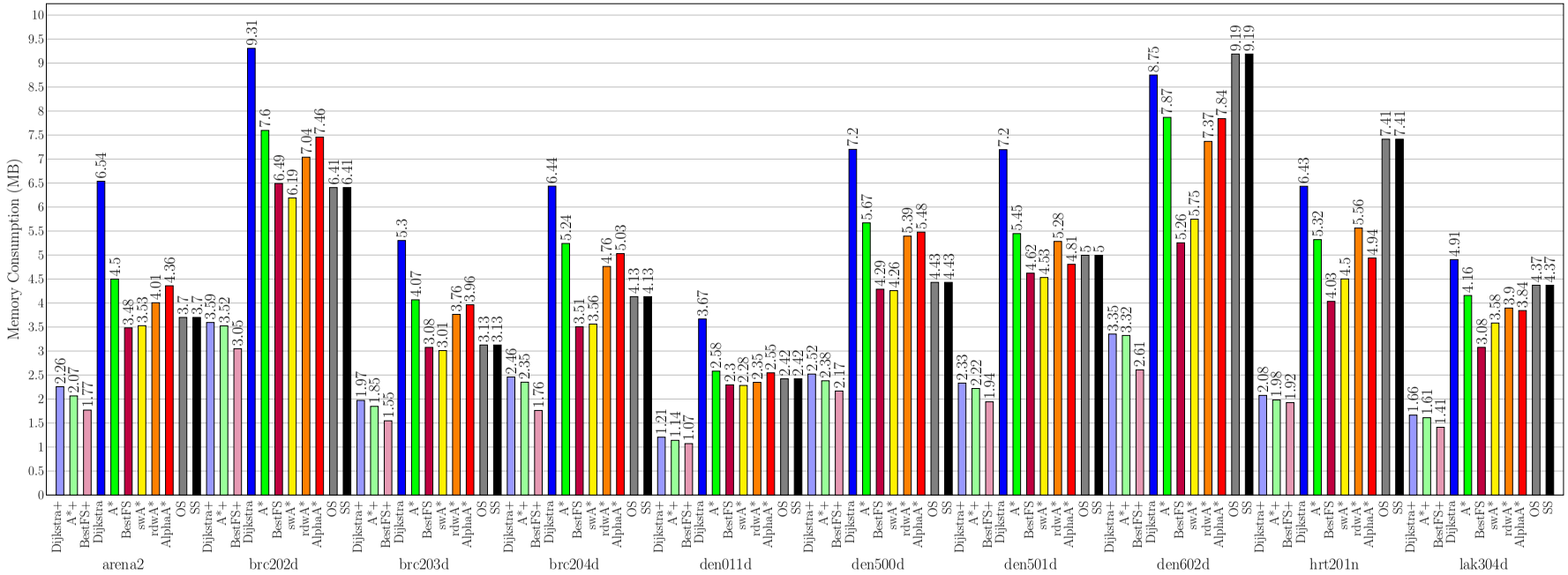


Figure 3.14: *Dragon Age: Origins* memory consumption results (test 2).

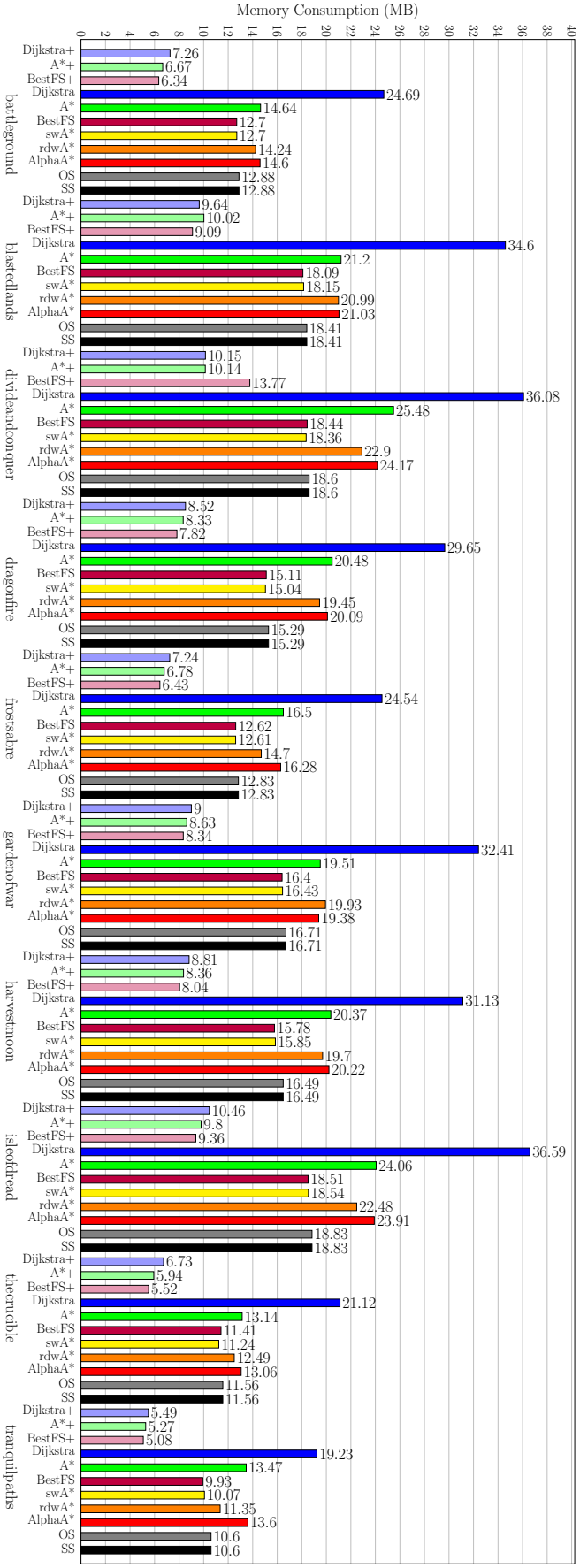


Figure 3.15: *Warcraft 3*'s memory consumption results (test 2).

set (if they are not therein yet). The closed set accumulates nodes already processed, while the open set has a more dynamic behavior in the sense that its size changes over time in a non-monotonic way by accumulation and retrieval operations of nodes.

In Figs. 3.12 and 3.13 the memory space occupancy values are presented for each map for either pathfinder used in `Test 1`, while for `Test 2` these results are illustrated in Figs. 3.14 and 3.15.

After a brief analysis of Figs. 3.12, 3.13, 3.14, and 3.15, we notice that:

- The consumption of memory tends to increase with the number of passable nodes. In practice, larger space search means that more memory consumption.
- The total consumption of memory is less for influence-based pathfinders (Dijkstra+, A*+, and BestFS+) than for their homologous pathfinders without influence (Dijkstra, A*, and BestFS). The memory consumption is lower for an influence-based finder that for its homologous finder without influence because attractors and repellers impose constraints to the game space search, even considering that the resulting path is certainly longer. Thus, a reduced game space search translates into less memory consumption.
- The general purpose bounded searches that required less memory (varying per map) usually are close to the lowest consuming influence-based pathfinder (either Dijkstra+, A*+, or BestFS+) in `Test 1`. The same does not apply to `Test 2` where influence-based pathfinders require less memory.

3.4.8 Time Performance

It was expected that the time performance would depend on the number of iterations carried out by each pathfinder. Each pathfinder's iteration picks up a node from the open set and turns it into a closed node. This means that the number of iterations is equal or greater than the number of closed nodes.

In fact, after a brief glance at Figs. 3.16, 3.17, 3.18, and 3.19, we see that both influence-based pathfinders (Dijkstra+, A*+, and BestFS+) perform better than their counterparts without influence. This can be explained by the smaller number of iterations required by influence-based pathfinders to find the way from the start node to goal node.

Moreover, these results also indicate that influence-based pathfinders outperform their pathfinders without influence, in spite of the mean time per iteration is longer for the former ones than for the latter ones.

In fact, taking into account Eqs. 3.7, 3.9, and 3.11 each iteration of the influence-based pathfinder performs more computation if the node being processed is under the influence of a propagator. On the other hand, the goal-oriented placement of propagators imposes important constraints to the size of the search space. Consequently,

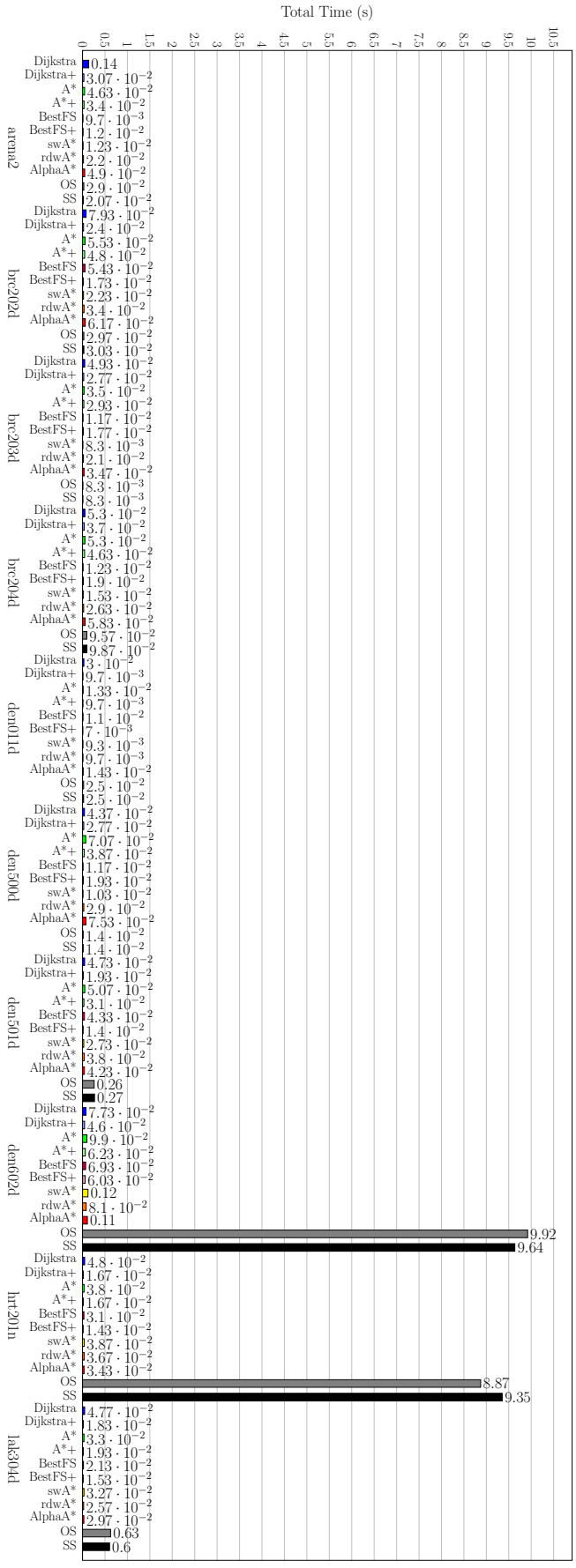


Figure 3.16: *Dragon Age: Origins* time performance results (test 1).

Figure 3.17: *Warcraft 3*'s time performance results (test 1).

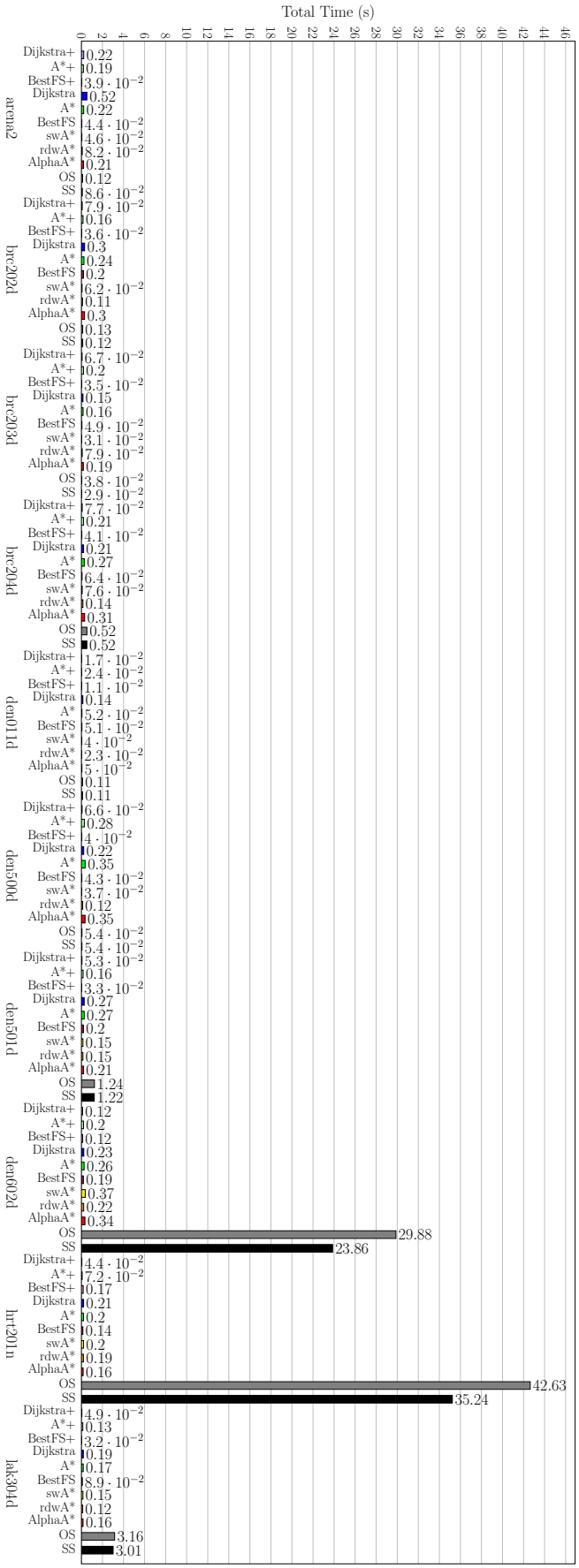


Figure 3.18: *Dragon Age: Origins* time performance results (test 2).

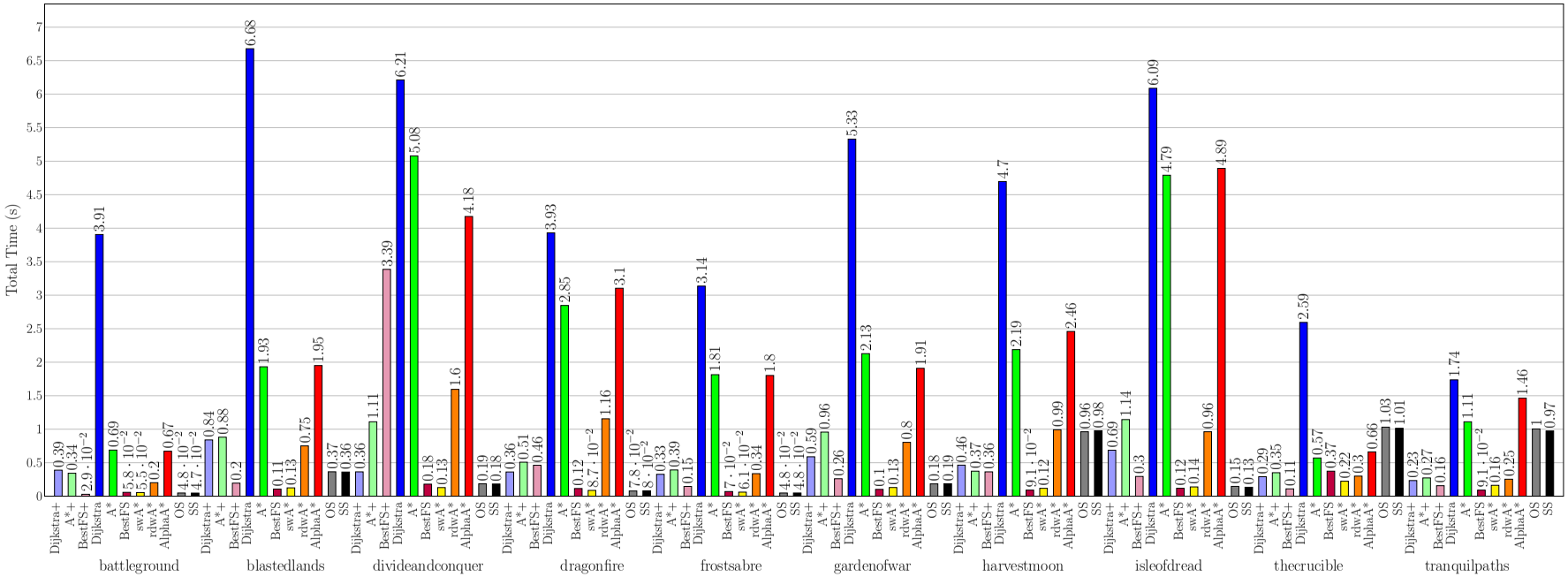


Figure 3.19: *Warcraft 3*'s time performance results (test 2).

the number of closed nodes (and iterations) reduces significantly relative to pathfinders without influence. This explains why influence-based pathfinders perform better than their counterparts without influence.

Comparing against general purpose bounded searches, at least one of the influence map-based pathfinders, most often BestFS+ beats the fastest general purpose bounded search (varies from map to map). This is consistent with either `Test 1` or `Test 2` results.

The reduction in time from the canonical to the influence-based version is more significant in Dijkstra, followed by A*, which is in turn followed by BestFS.

3.5 Further Discussion and Limitations

It was already discussed why influence-based pathfinders outperform their counterparts without influence, in terms of memory and time processing. Therefore, it remains to discuss how two out of the three problems listed in the beginning of the chapter have been solved using influence-based pathfinders, namely: path adaptivity and path smoothness.

3.5.1 Path Adaptivity

Traditional pathfinders assume that the scene is static, i.e., there are not objects moving across the virtual world, nor new objects coming in, nor objects getting out, nor objects being destroyed, and so forth. In other words, in dynamic scenes of game worlds, the graph of passable nodes changes over time, but usually this is taken into consideration only in adaptive pathfinders, which are often to processing or memory consuming to be employed in games. This problem was solved instead by placing propagators in the game world, wherever and whenever needed; e.g., a moving object in the scene can be associated to a repeller, so that no other entity clashes with it. Therefore, influence-based pathfinders easily adapt to changes in the game world, without having to restart the path search whenever someone adds or removes a propagator on the fly.

3.5.2 Path Smoothness

The problem of path smoothness has to do with how much natural (i.e., human-like) a moving object or NPC in the scene avoids obstacles. Thus, smoothing a path is not in the sense of more curvy, but a more human-like behavior when walking on the floor. In fact, by placing repellers and attractors in the game world, we can achieve a human-like behavior without additionally using the spline curves as in traditional pathfinders. Specifically, several cases can be made: a field-of-view influence map can be created, in which attractors are placed to create a sort of field of view graph which during path search will make paths much more human-like; attractors may be placed near specific

game positions we want to pass by when finding a path (e.g., a location with ammo to pass during a path search to grab additional ammo); repellers may be placed whenever a region is to be avoided, assuming that the affected nodes are not in the open set —if such was the case this would require traversing the open set at every iteration of the pathfinder, penalizing the proposed technique— may be done so during path search.

3.6 Further Remarks

In this chapter we proposed a technique that allows to integrate A* and its variants (Dijkstra's algorithm and best-first search) with influence maps. The proposed technique allows, while searching for a path, to avoid repellers and converge in the vicinity of attractors, if a path is to traverse nodes within the effect of said propagators. The proposed, technique is novel in the sense that: it uses Gaussian-based propagators; ensures that a repeller is avoided and an attractor converged to if there is an alternative, contrary to prior techniques. Also, the proposed technique allows to make adjustments to the path found during search, i.e., replacing yet permitting the same behavior of adaptive pathfinders without directly altering the search graph.

Chapter 4

Influence-centric A* Algorithms

In games, motion planning can be performed either using *pathfinders* (e.g., A*) or *fields* (potential fields or influence maps). Motion planners allow for an agent to traverse a game map between two locations (start and goal). Unfortunately, fields suffer from a common problem known as the trap of local extremum, i.e., an NPC may enter into a place from where it is no longer able to leave. To address this problem, Laue and Röfer [LR04] used a vector field for navigation of agents in a virtual world, resorting to a pathfinder algorithm only when an agent is in the vicinity of a local extremum to escape from it. This chapter describes an alternative novel solution to this problem, i.e., replacing the distance-based cost function of each canonical pathfinder (A*, Dijkstra, and best-first search) by an influence-based cost function that discards distances. Recall that in the previous chapter, we used a hybrid cost function that combines distance and influence values.

4.1 Introduction

Artificial intelligence in video games serves the purpose of making challenging and engaging games, as well as simulating seamlessly intelligent behavior when a player interacts with opponents, NPCs, enemies, and other entities. Among its many uses, artificial intelligence in games mainly deals with motion planning for NPCs [Nil98]. Motion planning for NPCs refers to two major techniques that address the problem of decomposing the movement of these entities, from one place to another in the game map, into a series of smaller discrete motions, namely: *path-finding* and *fields*.

4.1.1 Revisiting Pathfinders

In spite of the diversity of pathfinders in the literature, only a few can be considered as *adaptive pathfinders*. These pathfinders adapt to changes taking place in the game map and, consequently, in its search graph; for example, a bridge is destroyed (node removal), a crossing that connects two roads is added (node or edge addition), and so forth. To deal with search graph changes, adaptive pathfinders such as, for example, lifelong planning A* (LPA*) [KLF04] and D* (also known as dynamic A*) [Ste94, Ste95, KL05] introduce more data structures to handle node cost changes. Thus, *adaptive pathfinders* require more memory storage and I/O operations, as well as more node cost re-computations, what undermines their usefulness in games. There-

fore, the solution to deal with changes to the search graph (i.e., removal/addition of nodes from/to the graph) is recalculating the path from the start node to the goal node [Ste94, Ste95, KLF04, KL05][SKY08]. This explains why, at least partially, Dijkstra [Dij59] and A* [HNR68] pathfinders still are the standard pathfinders in games.

Also, A* and its variants produce piecewise linear paths, which results in jagged aesthetically unappealing paths [Rab00, Mar02, BMS04, Ger06, ASK15]. To make these jagged paths looking human-like, one has to smooth them out. Each jagged path is thus replaced by a curved path generated from an approximating spline. However, this solution is particularly computationally expensive, because it introduces many more computations to avoid collisions between an NPC —that moves along such curved path— and obstacles in the scene [Rab00].

Two solutions were proposed in the literature to deal with memory occupancy and time performance restrictions of A* and its variants (Dijkstra's and best-first search). The first consists in changing the value of the heuristic function (further discussed in detail in Section 4.3) to constrain the search space expansion, that is, to limit the size of open and closed lists. Pathfinders that adopt this solution are known as *general-purpose bounded suboptimal searches*, and include the static weighted A* [Poh70a], dynamic weighted A* [Poh73, TR09], AlphA* [Ree99], and optimistic and skeptical searches [TR08, TR10]. Pathfinders that adopt the second solution utterly discard such lists, and include the IDA* [Kor85] and fringe search [BEHS05]. In this chapter, we adopt the first solution because we also change the global cost function on nodes to restrain the search space expansion.

4.1.2 Revisiting Influence Fields

When using influence fields, the underlying idea is not to find a path between two locations, the start and end locations. Instead, such an idea is obstacle avoidance. That is, one intends to steer an entity (e.g., an NPC) to avoid collisions with obstacles while moving around the game world, which results in an human-like movement of NPCs since the influence function is smooth (or differentiable). Therefore, influence fields can be called steering fields. There are two types of steering fields: scalar influence fields (also known as *influence maps*) and vectorial influence fields (also known as *potential fields* or *flow fields*).

Artificial *potential fields* are a robotics technique [Kha85][Ark86][Ark87], which was later introduced into games by the hand of Thurau et al. [TBS04]. The core idea is to emulate the movement of an object within a magnetic field, which results from summing up subsidiary potential fields generated by propagators. A propagator is the center of each subsidiary potential field that gradually fades to zero [HJ08b]. There are two kinds of propagators: attractor, whose charge makes objects move towards it; repeller, whose charge makes objects move away from it. Propagators may be tied to static and dynamic game entities (e.g., a house as a static entity and a bot as a dynamic entity

that moves around) or not. For example, when steering an entity by simply considering other entities and obstacles as repellers, a moving entity will naturally avoid any static or moving obstacle.

On the other hand, *influence maps* were first used in the GO game [Zob69] to select the next tactical decision or GO move. That is, they were not initially used for motion planning of NPCs, partially because they suffer from a problem known as local extremum trap. Assuming that an NPC moves in the direction of decreasing values of the field, a local extremum is referred to as a local minimum; when it moves in the direction of increasing values of the field, it is referred to as local maximum. A local extremum is a map location where the field value of every single point in its neighborhood is either lower (local maximum) or higher (local minimum). Once at a local extremum, an NPC becomes trapped, being no longer able to leave [Goo00], as there is no place in its vicinity with either higher (when converging to higher field values) or lower (when converging to lower field values) field value to move.

4.1.3 Research Questions

Interestingly, some authors have used pathfinders to solve the problem of local minimum/maximum inherent to influence fields (see Laue and Röfer [LR04] for further details). In this case, the pathfinder is only used to bring an NPC out the influence of a minimum/maximum. That is, the pathfinder only acts locally in the neighborhood of each extremum. Therefore, path-finding is used to solve the local extremum trap problem of influence fields. In the present chapter, the approach is the opposite, because we use influence fields to solve the problems of pathfinding.

Therefore, the main research question (RQ) in this chapter is as follows:

> *Is it feasible to use influence-based costs instead of distance-based costs to solve the problems of canonical pathfinders?*

Note that our our proposal differs from prior works, as those due to Laue and Röfer [LR04], Millington [Mil06], and Paanakker [Paa08]. As mentioned above, Laue and Röfer use pathfinding to solve the problem of the local extremum of potential fields. That is, use a distance-based cost pathfinder to flee from a local extremum only when an agent gets entrapped therein.

Millington [Mil06] suggested altering the cost function of conventional pathfinders (Dijkstra and A*) by adding an extra term concerning the summation of tactical weighted decisions, but no implementation details were put forward. Paanakker [Paa08] developed the idea of changing the cost function due to Millington [Mil06], so that the extra term of the cost function takes into account propagators of constant influence within their radii of influence. More specifically, this latter solution adds a constant influence-valued term to the distance-based functions of Dijkstra and A*.

Instead, our solution is to replace distance-based costs of canonical pathfinders by influence-based costs. As a consequence, each resulting pathfinder will make an NPC avoid obstacles naturally. Also, the proposed technique in this chapter considers continuous Gaussian influence fields rather than discrete influence fields as those proposed by Millington [Mil06] and Paanakker [Paa08].

4.1.4 Contributions

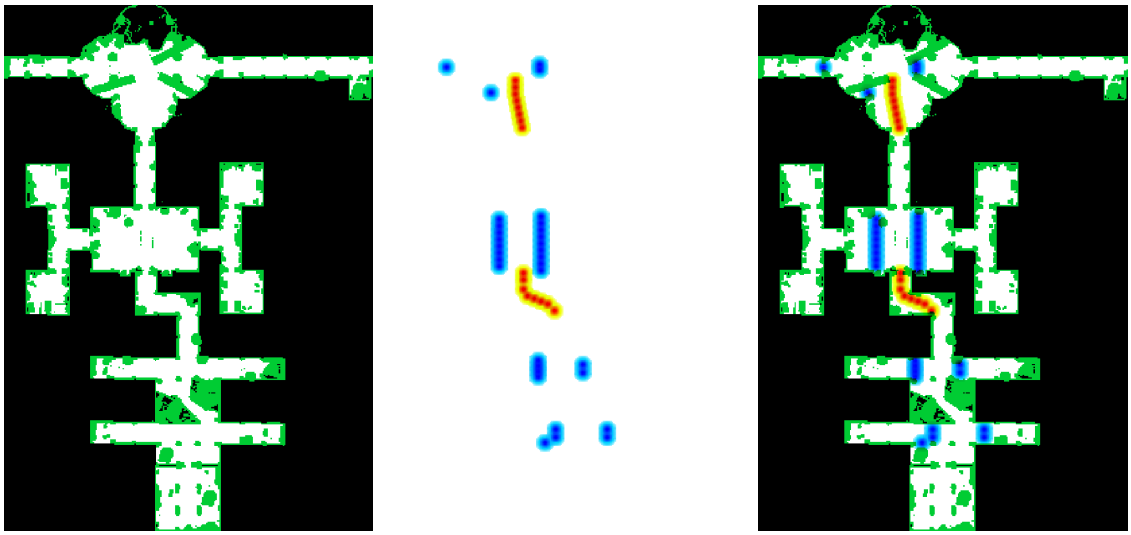
The main contribution of this chapter is a novel technique that allows us to re-design pathfinders built upon a width-based search strategy —as it is the case of A*, Dijkstra, and best-first search— to make them adaptive to changes in the game world. The adaptivity of these new pathfinders stems from their divergence from repellers (i.e., obstacle avoidance) and convergence to attractors. This adaptive behavior is similar to the technique introduced in the previous chapter, but instead of combining distance-based costs with influence-based costs, now we only use influence-based costs.

Consequently, we observed the following:

- These adaptive pathfinders remain complete so that the annoying problem of the local extremum vanishes.
- These adaptive pathfinders enjoy a preemptive avoidance of obstacles in the line of sight, which is much of an human-like behavior. To our best knowledge, only any-angle pathfinders enjoy such a human-like behavior [NDKF07, YBHS11].
- These adaptive pathfinders significantly reduce the search space, so that they require less memory consumption and likely a less number of computations; that is, they likely converge faster toward the goal location. This is particularly noticeable in adaptive Dijkstra and A* pathfinders. However, as we will see further ahead, the adaptive best-first search does not present any noticeable improvements in respect to memory space and time consumption because its search space is larger than the one of the canonical best-first search.

4.2 Revisited Theory of Fields

In our approach, an attractor is a local minimum of a single negative-charge scalar field, while a repeller is a local maximum of a single positive-charge scalar field. Intuitively, an *influence field* is the result from summing single-charge scalar fields of attractors and repellers placed on the game map. This is illustrated in Fig. 4.1, where attractors are depicted in red-to-yellow and repellers are in blue-to-cyan. For the sake of convenience, we consider each game map is tiled into congruent squares, i.e., a 2D grid. However, our approach applies to any search graph.



(a) (b) (c)

Figure 4.1: Representation of a grid-based game world: (a) game map; (b) influence map with attractors/repellers; (c) game map together with influence map.

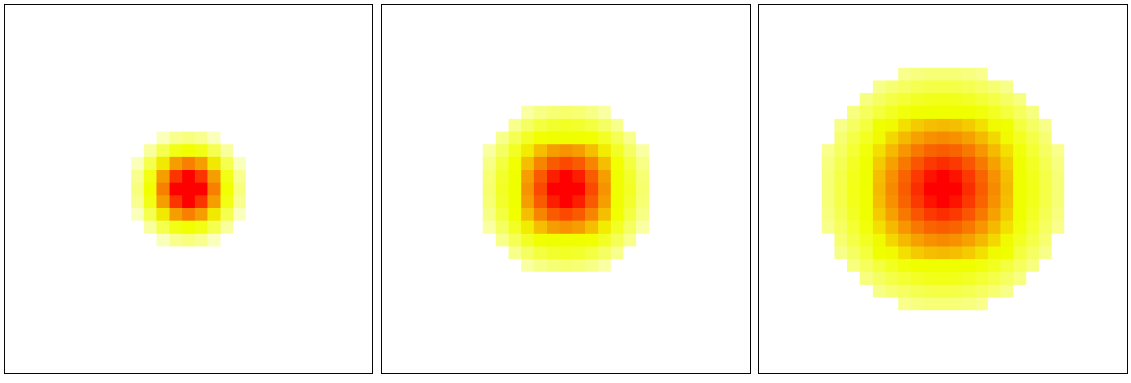
In the present chapter, an influence field is a real-valued Gaussian function of two variables [Par92], i.e., f is defined at every single point of $\mathbb{R}^2$. More specifically, the single-charge influence field generated by a repeller i is given by

$$f_i(\mathbf{p}) = a\, e^{-d_i^2 \cdot \delta_i^2} \tag{4.1}$$

with

$$a = \frac{1}{2\pi\sigma^2}, \qquad d_i = \|\mathbf{p} - \mathbf{p}_i\|, \qquad \delta_i = \frac{1}{\sqrt{2}\,\sigma} \tag{4.2}$$

where a is the amplitude of the Gaussian, d_i the distance from every point $\mathbf{p} \in \mathbb{R}^2$ to the center $\mathbf{p}_i$ of the repeller i, and δ_i the Gaussian decay factor with the distance to the center of the repeller i, and σ stands for the standard deviation. Fig. 4.2 shows us the bigger the decay, the lesser the influence area of a repeller.



(a) (b) (c)

Figure 4.2: Different values for the decay δ of an attractor: (a) with $\sigma = \sqrt{5}$; (b) with $\sigma = \sqrt{10}$; (c) with $\sigma = \sqrt{20}$.

Analogously, the single-charge influence field generated by an attractor j is as follows:

$$g_j(\mathbf{p}) = -a\, e^{-d_j^2 \cdot \delta_j^2} \quad \text{with} \quad d_j = \|\mathbf{p} - \mathbf{p}_j\|. \tag{4.3}$$

The overall scalar field generated by all n repellers and m atractors is as follows

$$I(\mathbf{p}) = \sum_{i=0}^{n-1} f_i(\mathbf{p}) + \sum_{j=0}^{m-1} g_j(\mathbf{p}) \tag{4.4}$$

In Fig. 4.1, we have 12 attractors in red and 13 repellers in blue. Note that the Gaussian function kernel associated to each attractor/repeller is unbounded, i.e., it is non-zero, yet it tends to zero, at any point of the game map. To speed up computations, in particular when an attractor or repeller is moving on the game map, we use a truncated Gaussian kernel, i.e., we consider the value of the function goes to zero for a small threshold $\tau \leq 0.1$. This allows us to determine the influence radius d_i of each i-th repeller (the same for an attractor) as follows:

$$\tau = a\, e^{-d_i^2 \cdot \delta_i^2} \tag{4.5}$$

or, equivalently,

$$d_i = \frac{1}{\delta}\sqrt{\ln(\frac{a}{\tau})} \tag{4.6}$$

Assuming that Δ represents the size of each grid tile, the expression $2\lceil\frac{d_i}{\Delta}\rceil \times 2\lceil\frac{d_i}{\Delta}\rceil$ represents the square influence neighborhood of the repeller i centered at $\mathbf{p}_i$, where $\lceil\frac{d_i}{\Delta}\rceil$ stands for the smallest integer not less than $\frac{d_i}{\Delta}$.

Recall that the leading idea of our motion planners is to blend a canonical pathfinder with a Gaussian influence field, so that we end up having a pathfinder that makes a moving entity to avoid obstacles in its way to the goal, without being trapped by minima. This has been accomplished by replacing the canonical distance-based cost function by the aforementioned influence-based cost function I (cf. Eq. (4.4)). In this formulation, we have assumed the game entity is moving in the direction of decreasing values of the field I.

4.3 Influence-centric Pathfinders

Let us now see how the distance-based cost function $F(n)$ of canonical pathfinders (i.e., A* and its variants) gives place to an influence-centric cost function $F(n)_+$. Specifically, we are interested in knowing how the two sub-functions of $F(n)$, that is, the distance-so-far $G(n)$ and the heuristic $H(n)$, change when one replaces distances by influence values.

4.3.1 Influence-centric A* Pathfinder

A* cost function is as follows:

$$F(n) = G(n) + H(n) \tag{4.7}$$

where $G(n)$ is the cost so far to (CSF), that is, the cost from the start node to the current node n, and $H(n)$ is the heuristic which denotes the cost estimate from the current node n to the goal node. When one uses influences instead of distances, Eq. (4.7) takes the following form:

$$F^{+}(n) = G^{+}(n) + H^{+}(n) \tag{4.8}$$

where $G^{+}(n)$ is the influence-centric CSF value and $H^{+}(n)$ is the influence-centric heuristic value at the node n. More specifically, $G^{+}(n)$ is as follows:

$$G^{+}(n) = \begin{cases} I_e(n) + I_{neutral} & , I_e(n) < -\tau \\ I_{neutral} & , I_e(n) \in]-\tau, \tau[\\ [I_e(n) + I_{neutral}] \ . \ u & , I_e(n) > \tau \end{cases} \tag{4.9}$$

where the topmost branch concerns the CSF values at cells under the influence of any attractor, while the bottommost branch concerns repelling influences; the middle branch denotes the CSF for cells that are not under influence of any attractor or repeller, i.e., neutral influences; $u = N - v$ is the number of the unvisited nodes, with N denoting the number of traversable nodes, and v the already visited nodes. Note that a visited node is an evaluated node or whose cost has been calculated at least once even if not placed in the *open set*. The threshold value τ (cf. Eq. (4.5)) determines the interval $]-\tau, \tau[$ of neutral influence, i.e., where attractors and repellers do not exert any influence. The value of $I_{neutral}$ refers to the neutral influence value, which must always be above zero, and is given by:

$$I_{neutral} = \begin{cases} \tau & , n_A = 0 \\ |\mathsf{min}(I_A)| & , n_A \neq 0 \end{cases} \tag{4.10}$$

where n_A is the number of attractors in the influence map and $\mathsf{min}(I_A)$ the minimum attractive influence in the entire influence field. In turn, the influence value $I_e(n)$ of an edge connecting a node n to one of its neighbors $n+1$ is given by:

$$I_e(n) = \frac{I(n) + I(n+1)}{2} \tag{4.11}$$

where $I(n)$ is the influence value at node n and $I(n+1)$ the influence value at one of its neighbors $n+1$.

The value of $I_{neutral}$ adds to topmost and bottommost branches of Eq. (4.9) to ensure all costs are positive, as A* and its variants only work with positive costs, i.e., influence values are shifted to an interval in which all influences are positive. Furthermore, in the bottom most branch of Eq. (4.9), one multiplies the shifted value of $[I_e(n) + I_{neutral}]$ by u to ensure that nodes (in the *open set*) under repelling influences are solely evaluated when there are no viable lower cost alternatives, i.e., nodes under repelling influences will be avoided as long as no other node with lower CSF influence value exists. This will result in the avoidance of paths near repelling influences as long as a path exists to the goal node that does not require forcing a path trough a repeller, i.e., graph expansion must consider a global influence at a node, instead of the local influence of a repeller at the same node.

In regards to the heuristic function, we have:

$$H^{+}(n) = \begin{cases} I_{min} + I_{neutral} & , I_{min} < -\tau \\ I_{neutral} \cdot \mathsf{dist}(n) & , I_{min} \in]-\tau, \tau[\\ [I_{min} + I_{neutral}] \ . u & , I_{min} > \tau \end{cases} \tag{4.12}$$

where $I_{min} = \mathsf{min}(I(n), I(n+1))$ is the minimum influence value among all neighbors $(n+1)$ of node n and the node n itself, and $\mathsf{dist}(n)$ is the Euclidean distance from n to goal node. The topmost branch of Eq. (4.12) is to calculate the cost estimate for cells under attractive influences, the middle branch for neutral influences, while bottommost branch is for repelling influences.

Figs. 4.3(a) to (h) illustrate the behavior of the A* pathfinder either in the absence and presence of propagators, where the path is in dark gray, visited nodes are in light blue, attractors are in red (near the center) to yellow (as their attraction decreases), and repellers in dark blue (near the center) to light blue (as its repulsion decreases). The influence-centric A* pathfinder is here called α^{+} pathfinder.

It is important to refer that in the canonical pathfinders like A*, Dijkstra's and best-first search, the search only terminates when the goal node is found, or the set of nodes to be evaluated (*open set*) is empty. In our adaptive pathfinders, the influence field is only updated if a propagator either moves, is removed, or is added. But, the costs of the nodes already in the *open set* are not altered. That is, changes to the influence field will solely result in changes to the path being determined in yet to be evaluated nodes. These changes could be addressed by checking for all nodes in the *open set* if there were changes in their cost. However, this would expectedly penalize the influence-based pathfinder performance. The reason for such is that, instead of accessing the first node (in the implemented source code) of the priority queue (*open set*), one would have to traverse all the open set, to check if there were nodes whose costs had to be altered at each pathfinder iteration.

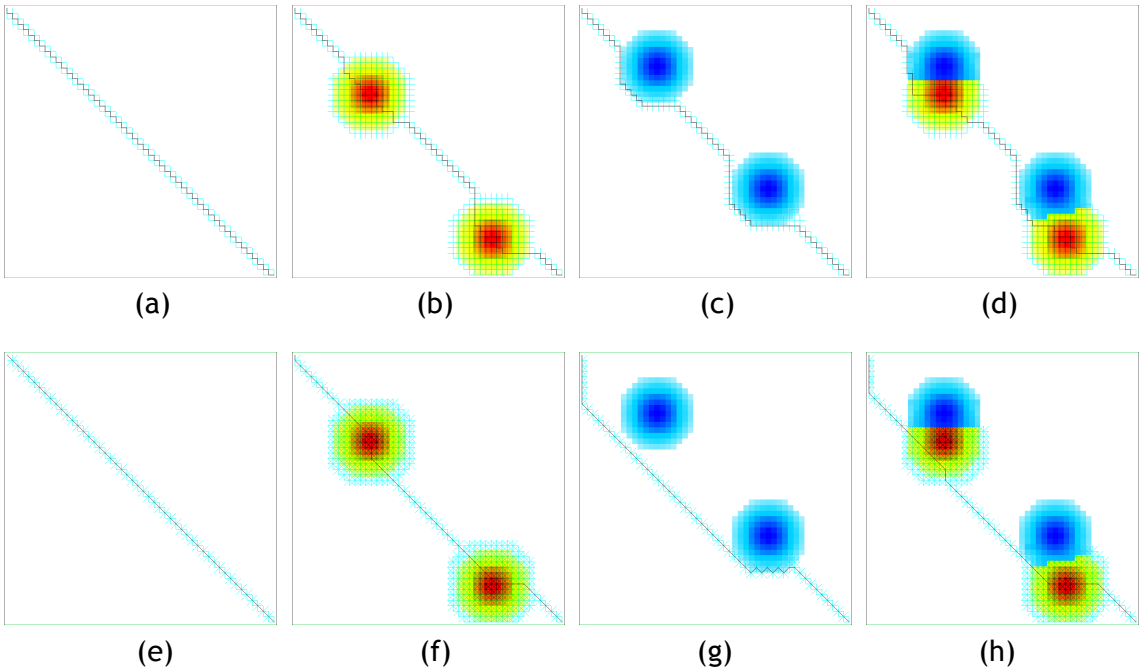


Figure 4.3: The α^+ algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

4.3.2 Influence-centric Dijkstra Pathfinder

The influence-centric Dijkstra pathfinder is here called δ^+ pathfinder. Dijkstra's algorithm is a variant of A* whose heuristic function is zero. Consequently, Dijkstra's explores the search graph according to a breadth-first search pattern. Thus, with $H^+(n) = 0$, Eq. (4.8) turns into the following:

$$F^+(n) = G^+(n) \tag{4.13}$$

where $G(n)_+$ is the CSF function as given by Eq. (4.9). Figs. 4.4(a) to (h) illustrate the behavior of the δ^+ algorithm pathfinder either in the absence and presence of propagators.

4.3.3 Influence-centric Best-First Search Pathfinder

The influence-centric best-first search pathfinder is here called β^+ pathfinder. The best-first search algorithm is a variant of A* whose CSF function is zero. Consequently, best-first search explores the search space according to a depth-first search pattern.

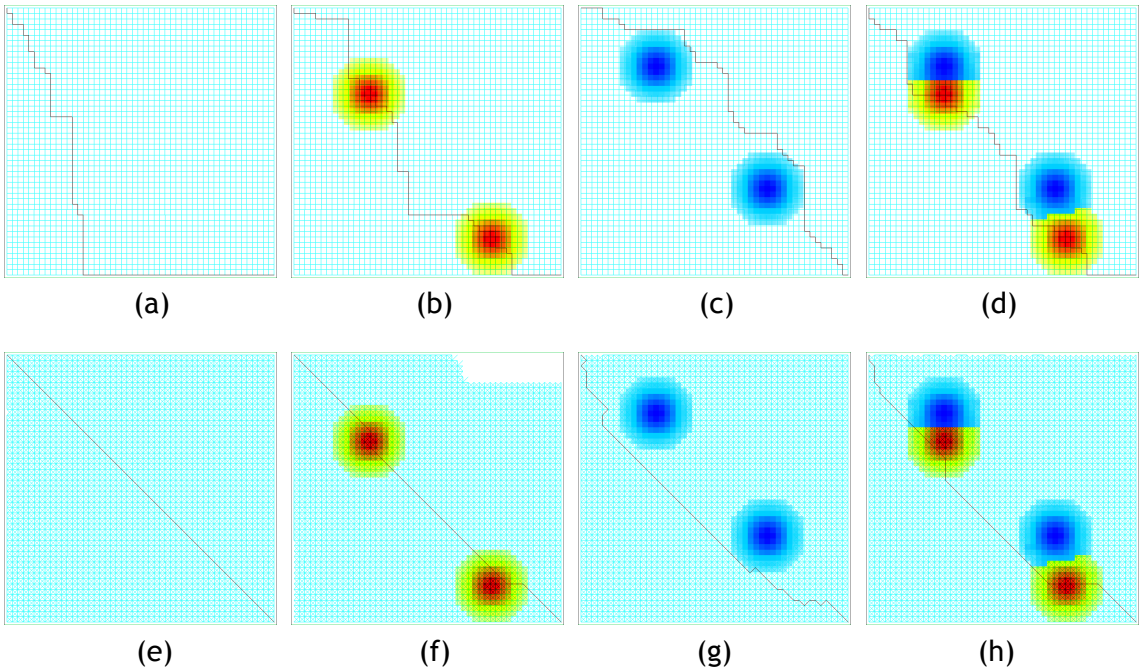


Figure 4.4: The δ^+ algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

Thus, with $G^+(n) = 0$, Eq. (4.8) becomes the following:

$$F^+(n) = H^+(n) \tag{4.14}$$

where $H^+(n)$ is the heuristic function as given by Eq. (4.12). Figs. 4.5(a) to (h) show the behavior of the β^+ algorithm pathfinder either in the absence and presence of propagators.

4.4 Experimental Results

In this section, we carry out a comparative analysis between Dijkstra (point-to-point variant), A*, best-first search, δ^+, α^+, and β^+ and five general purpose suboptimal bounded search algorithms, namely, static weighted A* (swA*), revised dynamic weighted A* (rdwA*), AlphA*, and optimistic and skeptical searches (OS and SS), using a bound value of 1.5 as in [TR10].

The comparative analysis was performed taking into account two testing scenarios. The first was set up without changing the graph at all. The second was set up by changing the graph search to test the capability of each pathfinder of handling adaptive scenarios.

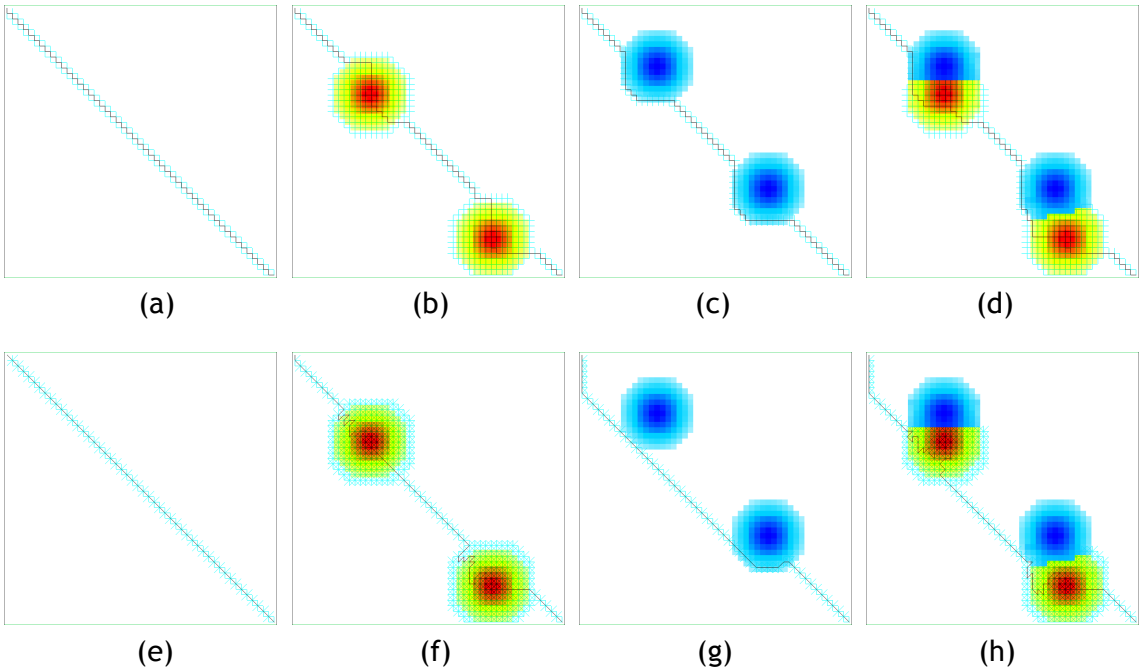


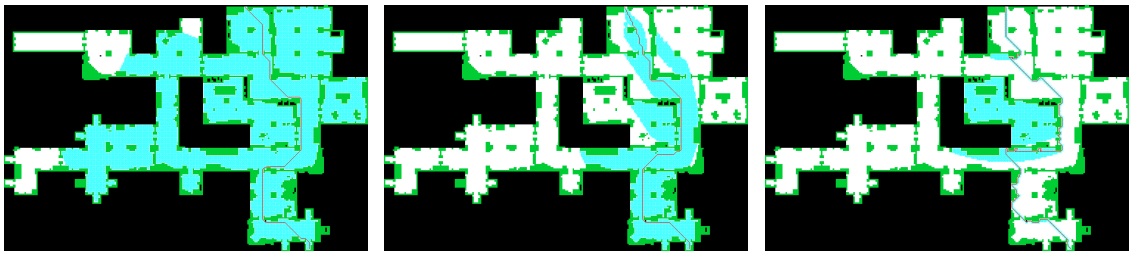
Figure 4.5: The β^+ algorithm for a 50×50 grid (i.e., a graph with 2500 nodes): (a) without diagonals or influence map; (b) without diagonals and with influence map generated by 2 attractors; (c) without diagonals and with influence map generated by 2 repellers; (d) without diagonals and with influence map generated by 2 attractors and 2 repellers; (e) with diagonals and without influence map; (f) with diagonals and with influence map generated by 2 attractors; (g) with diagonals and with influence map generated by 2 repellers; (h) with diagonals and with influence map generated by 2 attractors and 2 repellers.

4.4.1 Software/Hardware Setup

In total, eleven pathfinders were implemented in Java programming language. The tests were performed on a desktop computer running a Windows 7 64-bit Professional operating system, equipped with an Intel Core i7 920 @ 2.67GHz processor processor, 8 GB Triple-Channel DDR3 de RAM, and a NVIDIA GeForce GTX 295 graphics card with 896MB GDDR3 RAM.

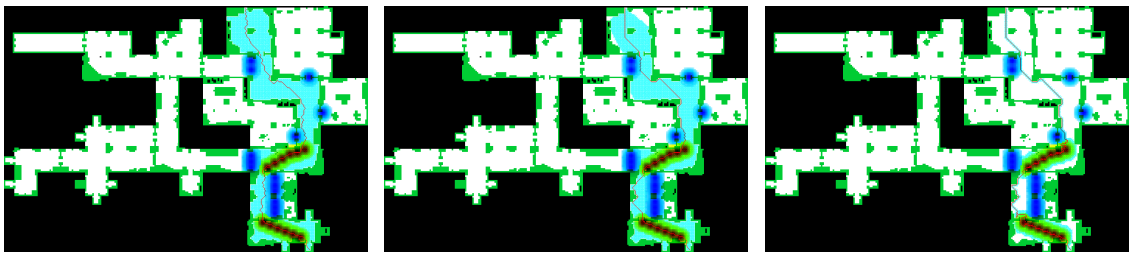
4.4.2 HOG Map Dataset

In testing, we used a dataset of 20 game maps taken from the HOG2 map repository (`http://movingai.com/benchmarks`); more specifically, we used 10 *Dragon Age: Origins* maps, as well as 10 *Warcraft 3* maps. *Dragon Age: Origins* is a role playing game (RPG) mostly consisting of indoor dungeon-like scenarios, while *Warcraft 3* is a real-time strategy (RTS) game mainly comprising outdoor scenarios, including swamps or islands. Unfortunately, HOG repository is empty of first-person shooter (FPS) maps. As an example, Fig. 4.6 shows paths (and search expansion) generated by A*, Dijkstra's, and best-first search pathfinders (without influence), while Fig. 4.7 shows paths generated by α^+, δ^+, and β^+ pathfinders (with influence of attractors and repellers).



(a) Dijkstra (b) A* (c) best-first search

Figure 4.6: Identical paths found by Dijkstra's, A*, and best-first search algorithms (without influence of attractors and repellers) within the *Den011d* map of *Dragon Age: Origins*.



(a) δ^+ (b) α^+ (c) β^+

Figure 4.7: Distinct paths found by Dijkstra's, A*, and best-first search algorithms (with influence of attractors and repellers) within the *Den011d* map of *Dragon Age: Origins*.

4.4.3 Grid-based Game Map

Both *Dragon Age: Origins* and *Warcraft 3* build upon grid-based maps, i.e., each map consists of square tiles or cells. Each tile and its eight neighbor tiles form a 3×3 pattern unless it is a boundary tile of the map. Thus, a map can be implemented as a 2-dimensional array of size $x \times y$, where x represents the number of tiles in x direction, while y denotes the number of tiles in y direction.

In regards to tiles, they can be categorized into two sets: passable and impassable. There are three sorts of impassable tiles: black, blue, and green. As illustrated in Fig. 4.7, black tiles are those out-of-bounds of the map and can be identified as the black tiles of the map. Green tiles represent walls and plants, as well as other decorative elements in indoor scenarios, as those of *Dragon Age: Origins*. In regards to outdoor scenarios as of *Warcraft 3*, green tiles represent forests and other obstacles. In turn, blue tiles correspond to deep water, so they only exist in outdoor scenarios.

As far as passable tiles are concerned, we can say that there are two sorts of passable tiles: white and blue sapphire. White tiles are the most common in both indoor and outdoor scenarios. Blue sapphire tiles are only found in outdoor scenarios, but they have higher costs because they denote shallow water of lakes, rivers, and oceans. Additionally, to show how large the search graph expansion is, the visited tiles are represented in cyan for the sake of legibility, as illustrated in Fig. 4.7.

4.4.4 Grid-based Influence Maps

We used an influence map for each game map as needed for motion planning; they are of the same size $x \times y$. As shown in Fig. 4.5, red-to-yellow nodes are those under influence of an attractor, while blue-to-cyan nodes correspond to those under influence of a repeller. For simplicity, tests were performed using attractors defined by the parameters $\sigma = -\sqrt{10}$ and $\tau = 0.1$, while repellers were parameterized through $\sigma = \sqrt{10}$ and $\tau = 0.1$ (see Eqs. (4.2) and (4.5)).

Attractors and repellers were empirically placed on the map. In more specific terms, the placement of attractors is performed using the line of sight to goal node when an imaginary agent travels across the map. In regards to repellers, they are placed on the map to constrain the search space, preferably in narrow passages to larger areas where the goal node surely is not there (see Fig. 4.7).

4.4.5 Testing Methodology

The methodology followed in testing was designed to show the advantages of combining A* pathfinders and influence fields, particularly in mitigating and solving the following issues: memory consumption performance, time performance, and path adaptivity. We adopted a testing methodology based on two aspects: (i) constrained search domain; (ii) a limited number of testing paths.

4.4.5.1 Constrained search domain

The search domain is constrained by the repellers placed on the map. However, a repeller does not prevent the passage to an agent; it only delays its traversal to another side. This is so because repellers are associated with the lowest priority in the process of searching over the graph. Thus, a repeller node is only traversed when there is no other node with higher priority in the open set.

4.4.5.2 Number of testing paths

We only use three paths per map, namely: (i) the potentially longest left (LPL) path, which starts at the right bottom most node and terminates at the left topmost node; (ii) the path between two random attractors, but, for maps without attractors, we select their homologous tiles or nodes; (iii) the path between any other two random attractors. These paths are chosen within the search domain delimited by repellers; otherwise, the agent moving toward the goal node may get lost in some alleyway, so incurring in consumption penalties regarding memory and time. This also explains why we do not use paths between repellers, or between an attractor and repeller, as those penalties increase when graph search overflows the sub-graph delimited by repellers.

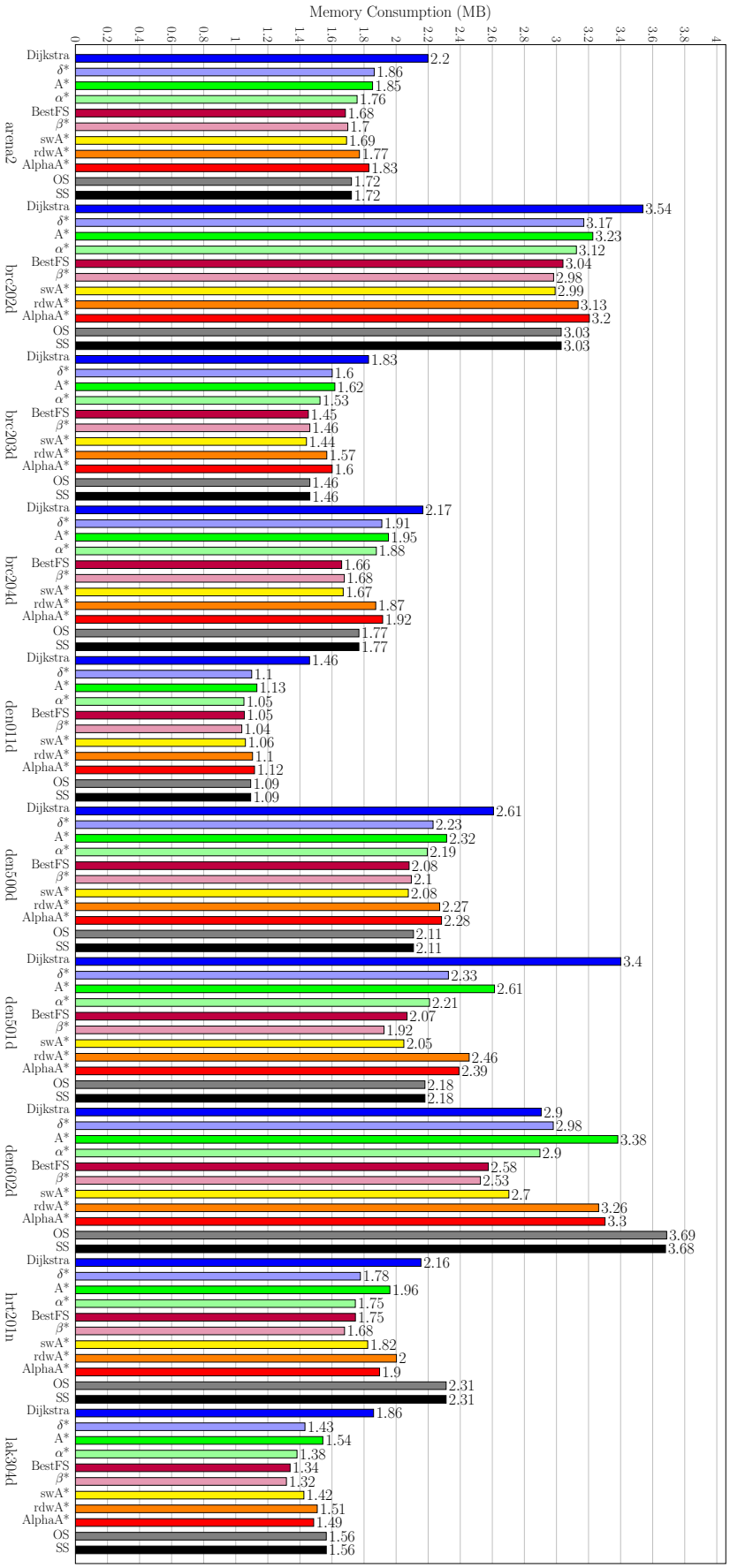


Figure 4.8: Memory space consumption for 10 maps of Dragon Age: Origins.

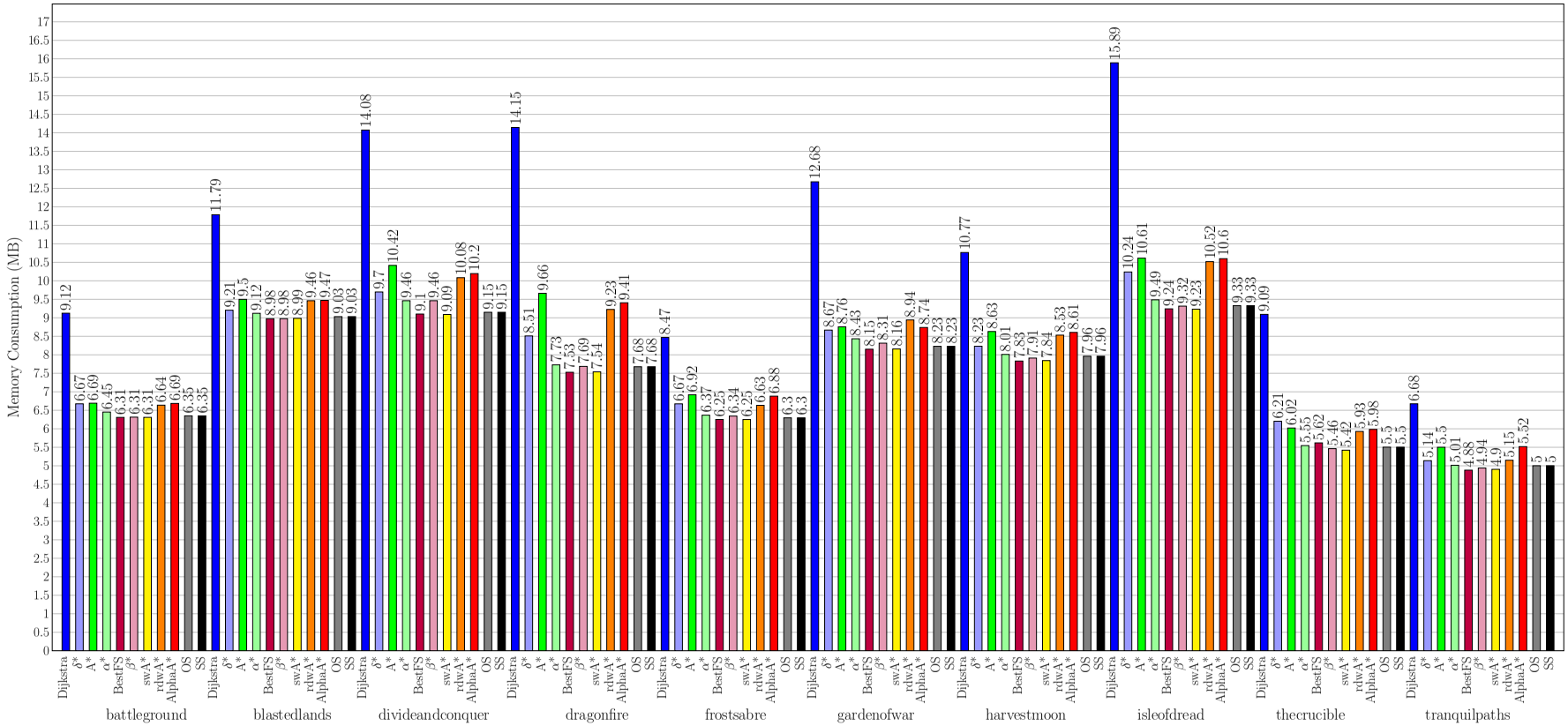


Figure 4.9: Memory space consumption for 10 maps of the *Warcraft 3*.

4.4.6 Memory consumption

Memory consumption is related to how extensive is the search across the graph. In fact, more graph nodes spanned by the search procedure means more memory consumption. Therefore, constraining the graph search is the most common strategy to consume less memory space. Hence, the development of bounded-search pathfinders, including those three A* variants proposed in this chapter. From Figs. 4.8 and 4.9, we rapidly come to the conclusion that *Warcraft 3* (W3) involves more memory consumption than *Dragon Age: Origins* (DAO), mainly because each DAO map is larger than the largest W3 map. We also see that Dijkstra pathfinder consumes more memory space than any other pathfinder, but it is hard to know which pathfinder is the best regarding memory consumption. For that purpose, we use the joint metric $\bar{x} + \sigma$, where $\bar{x}$ stands for the arithmetic mean time, and σ the standard deviation, of memory space consumption.

Table 4.1: Statistical memory results for *Dragon Age: Origins* maps.

Algorithm	$\bar{x}$	σ	$\bar{x} + \sigma$	Rank
Dijkstra	2.412035	0.687487	3.099522	11
δ^+	2.039070	0.653564	2.692634	5
A*	2.159688	0.728099	2.887787	10
α^+	1.976142	0.649041	2.625183	4
BestFS	1.869986	0.593313	2.463299	2
β^+	1.839690	0.575522	2.415212	1
swA*	1.892926	0.590100	2.483026	3
rdwA*	2.095207	0.697061	2.792268	6
AlphaA*	2.102999	0.709223	2.812223	7
OS	2.092551	0.774273	2.866824	9
SS	2.091829	0.772626	2.864455	8

Table 4.2: Statistical memory results for *Warcraft 3* maps.

Algorithm	$\bar{x}$	σ	$\bar{x} + \sigma$	Rank
Dijkstra	11.270334	2.949815	14.220150	11
δ^+	7.924547	1.665886	9.590433	7
A*	8.270875	1.854663	10.125538	10
α^+	7.562183	1.631015	9.193199	6
BestFS	7.389197	1.547743	8.936940	1
β^+	7.472680	1.620299	9.092980	5
swA*	7.373641	1.569414	8.943055	2
rdwA*	8.111414	1.870850	9.982264	8
AlphaA*	8.208440	1.807096	10.015537	9
OS	7.452737	1.569073	9.021810	3
SS	7.452737	1.569073	9.021810	4

DAO maps. From Table 4.1, we note that influence-centric pathfinders (i.e., α^+, δ^+, and β^+) consume less memory space than their canonical pathfinding counterparts (i.e., A*, Dijkstra, and BestFS). In general, they also consume less memory space than other

bounded-search pathfinders (i.e., swA*, rdwA*, and OS). In DAO maps, β^+ ranks first, while BestFS and swA* rank second and third, respectively, and this is valid not only on average ($\bar{x}$), but also overall ($\bar{x} + \sigma$).

W3 maps. Looking at Table 4.2, we see Dijkstra, A*, and AlphaA* perform worse than any other pathfinder in terms of memory occupation, but the remaining pathfinders consume the same memory space approximately, yet BestFS ranks first, because its heuristic-based search is steered toward the goal node, so it tends to evaluate fewer graph nodes.

Summing up, Dijkstra, A*, and Alpha* are the worst pathfinders regarding memory space consumption, while the remaining pathfinders consume similar amounts of memory space in DAO and W3.

4.4.7 Time performance

The time performance essentially depends on the number of (closed) nodes processed during the expansion of the graph search. A brief glance at charts shown in Figs. 4.10 and 4.11 allow us to easily conclude that OS and Dijkstra are the time-worst pathfinders for *Dragon Age: Origins* and *Warcraft 3*, respectively.

In regards to the time-best pathfinder, it is rather difficult to say which pathfinder is the quickest, unless we carry out a statistical analysis of the time performance of those nine pathfinders based on a metric that combines the mean time ($\bar{x}$) and its standard deviation (σ).

Table 4.3: Statistical time results for *Dragon Age: Origins* maps.

Algorithm	$\bar{x}$	σ	$\bar{x} + \sigma$	Rank
Dijkstra	0.077833	0.039801	0.117634	9
δ^+	0.032967	0.014824	0.047790	1
A*	0.061100	0.027673	0.088773	6
α^+	0.059467	0.036617	0.096084	7
BestFS	0.034467	0.027375	0.061842	4
β^+	0.034700	0.023447	0.058147	2
swA*	0.038267	0.039914	0.078180	5
rdwA*	0.037733	0.021892	0.059626	3
AlphaA*	0.064700	0.032480	0.097180	8
OS	2.395400	4.730486	7.125886	11
SS	1.739267	3.389504	5.128771	10

DAO maps. From Table 4.3, we observe that the three quickest methods are BestFS, δ^+, and β^+ on average (see column of $\bar{x}$); this time performance ranking also holds overall (see column 'Rank') because joint metric ($\bar{x} + \sigma$) maintains their relative positions. Furthermore, except OS pathfinder, there are no significant time differences between the benchmarked pathfinders because they all perform in less of 1/10 seconds. In short, influence-centric pathfinders perform well in indoor games as of DAO.

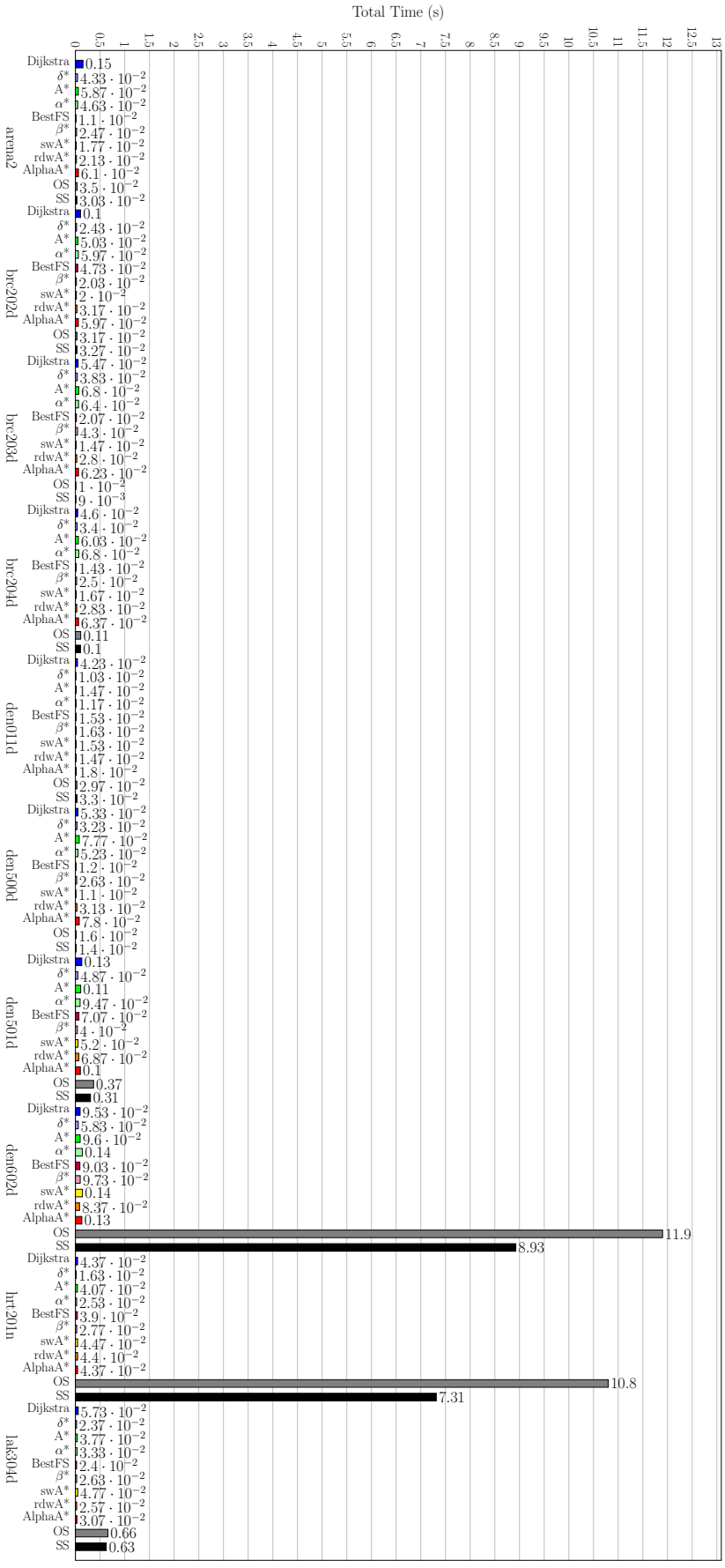


Figure 4.10: Time performance for 10 maps of Dragon Age: Origins.

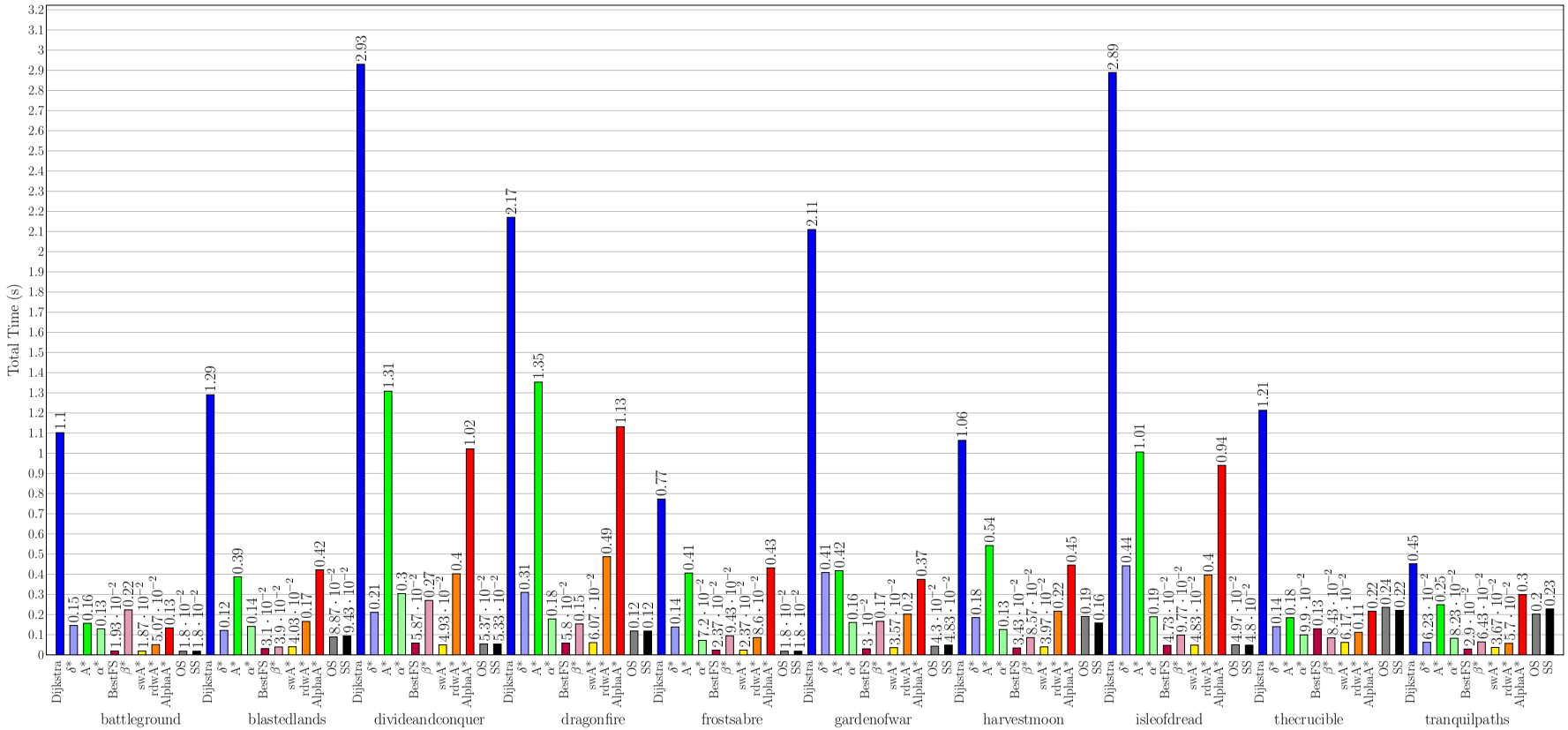


Figure 4.11: Time performance for 10 maps of *Warcraft 3*.

Table 4.4: Statistical time results for *Warcraft III* maps.

Algorithm	$\bar{x}$	σ	$\bar{x}+\sigma$	Rank
Dijkstra	1.599367	0.869132	2.468499	11
δ^+	0.216300	0.127863	0.344163	7
A*	0.601033	0.452838	1.053871	10
α^+	0.148033	0.067039	0.215072	6
BestFS	0.046100	0.032356	0.078456	2
β^+	0.127833	0.073854	0.201688	5
swA*	0.041467	0.014078	0.055544	1
rdwA*	0.217567	0.157663	0.375230	8
AlphaA*	0.541667	0.354654	0.896321	9
OS	0.101733	0.080816	0.182550	4
SS	0.100467	0.078639	0.179106	3

W3 maps. Looking at Table 4.4, we observe that swA*, BestFS, and OS are the quickest pathfinders on average. In fact, swA* ranks first, BestFS ranks second, and OS ranks third, though α^+ performs close to OS. That is, we have got two bounded-based pathfinders on top three, but none of them is an influence-centric pathfinder.

Summing up, the time performance of a pathfinder depends on the type of map. δ^+ is the best bounded-search pathfinder for DAO indoor maps, yet it ranks second overall, while swA* is the best pathfinder for W3 outdoor maps. Another conclusion is that neither Dijkstra nor A* is the adequate options as pathfinders for indoor and outdoor games, but BestFS ranks in top four in both types of games. In general terms, except for β^+, influence-centric pathfinders perform better than canonical pathfinders. Furthermore, only BestFS and swA* perform at interactive rates in W3, i.e., under 41 ms (or above 24 frames per second), but in DAO also δ^+, β^+, swA*, and rdwA* perform in real-time.

4.4.8 Path Adaptivity

In dynamic environments, scenes change over time with some objects moving around, others being destroyed, and new objects just coming in from elsewhere, and so forth. This means that in some circumstances the search graph also changes over time. More specifically, when we are dealing with a dynamic game map, nodes of its graph may swap from passable to impassable, or vice-versa.

It happens that adaptive pathfinders (i.e., pathfinders that cope with environment changes) have not been successful in games, although they are common in robotics (e.g., D*[ASK15], i.e., an adaptive variant of A*). Adaptive pathfinders are not practically used in games because their time performance drops to a level that it is better to redo the search fully than using an adaptive pathfinder [All11]. On the other hand, the three influence-centric pathfinders (i.e., δ^+, α^+, and β^+) here described are adaptive, because they cope with game map changes over time, which translate into removal

/adding of new graph nodes, as well as removal /adding of repellers and attractors (see Fig. 4.12).

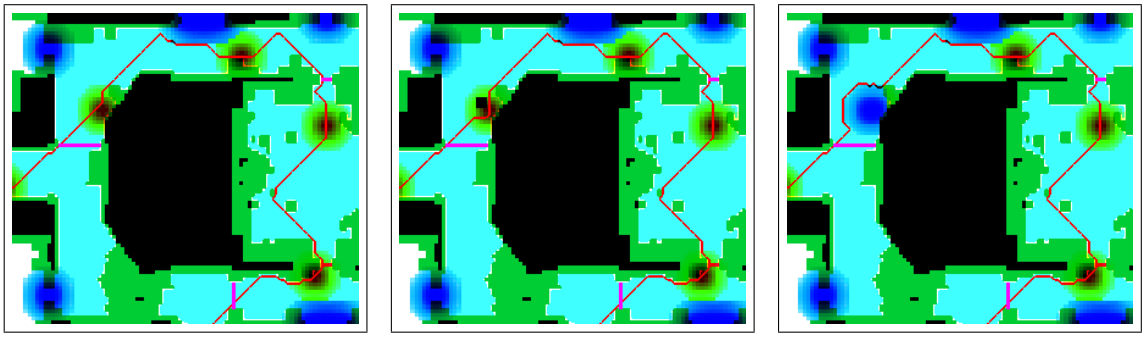


Figure 4.12: Adaptive pathfinding using α^+ for the *Hrt201n* map of *Dragon Age: Origins*: (a) without changing the influence field or search graph; (b) after removing a graph node and its 8 neighbors (in black); (c) after adding two repellers at the same location (in blue).

In general, a pathfinder consists of two stages: (i) forward discovery of a path from the start node to goal node; (ii) backward reconstruction of such path. Therefore, an environment change may occur in either pathfinder stage. If such change takes place during the forward discovery stage, no path correction needs to be done. But, if the environment change occurs during the backward reconstruction of the path, this path needs to be locally corrected to surround any removed passable node (e.g., a bridge blew up) or any repeller placed on the original path. The corrected part (or sub-path) of the original path is determined through the same pathfinder since experiments show us it incurs in an additional time that is proportional to the length of the corrected sub-path. Summing up, our influence-centric pathfinders adapt to environment changes, without the need to restart the pathfinder once again from the start toward the end.

4.5 Further Discussion

This chapter introduces three variants of Dijkstra [Dij59], A* [HNR68], and best-first search [Poh70b], which result from replacing traditional distance-based cost functions by influence-centric cost functions. Each influence-centric cost function expresses the placement and existence of attracting and repelling charges in the environment.

We have found three techniques in the literature that relate to ours somehow, as it is the case of those due to Laue and Röfer [LR04], Paanakker [Paa08], and Adaixo et al. [AAG15]. Laue and Röfer [LR04] take advantage of a vector field to perform standard vector field-based navigation of agents in a virtual world, using only a distance-based cost function pathfinder locally when an agent gets trapped at a local extremum. That is, the pathfinder is only used to release an agent from a local extremum, where it is trapped, not to find a path as is usual.

In regards to Paanakker [Paa08] and Adaixo et al. [AAG15], their works focus on integrat-

ing canonical pathfinders and influence maps. Basically, Paanakker [Paa08] modified the cost functions of Dijkstra's and A* pathfinders in order to include influence values associated to repellers and attractors, but such values are constant within the area of influence of each repeller/attractor, more specifically, -1 for attractors and +1 for repellers; consequently, the agent often ignores the presence of repellers and attractors, moving straight ahead through their influence areas. Furthermore, the human-like movement of the agent getting out from repellers and approaching attractors rarely happens and depends on the game map and tuning parameters, so it is not automated.

Adaixo et al. [AAG15] unsuccessfully attempted to solve the issue of the straight moving behavior of agents relative to repellers and attractors, as observed in Paanakker [Paa08], by replacing constant influence values by decreasing values obtained from a Gaussian kernel function. After a series of experiments, we noted that this happens because there is no guarantee that the cost function value of the next node to be evaluated is less than the cost function value of the current node. In contrast, we solved such a problem in our influence-based bounded-search pathfinders by ensuring that their cost functions monotonically decreases from the start node to the goal node.

It is worth noting that the main limitation of our technique lies in the non-automated placement of attractors/repellers in the game world. But, we could use a visibility graph to place attractors and repellers at reference locations to prevent an agent moving away from the right track, i.e., to prevent such agent getying in chambers and alleyways, so constraining the graph search expansion.

4.6 Further Remarks

In this chapter, we have proposed a technique that replaces the distance-based cost function of A*, Dijkstra and best-first search algorithms by an influence-centric cost function, which exclusively uses influence values instead distance values. This novel technique uses continuous Gaussian-based propagators so that it ensures that a moving object in the game world avoids repellers and passes through nearby attractors on its way toward the goal location. Furthermore, similar to the technique presented in the previous chapter, the technique out forward in this chapter allows to make adjustments to the path found during the search, i.e., the influence-centric technique allows for designing adaptive pathfinders.

Chapter 5

Best Neighbor First Search Algorithm

Path-finding refers to algorithms that find a path from a start node to a goal node of a search graph. A*, Dijkstra's and best-first search, and finge search algorithms are the most common pathfinders found in games. Dijkstra's, A* and fringe search algorithms ensure that an optimal path (if one exists) is found. Best-first search only ensures that a path is found, but it does ensures such path is optimal. However, best-first search has better time performance and consumes less memory space than Dijkstra's, A*, or fringe search. In this chapter, we introduce the *best neighbor first search* (BNFS) algorithm, which finds paths closer to optimal paths than best-first search and sometimes outperforms it in time performance.

5.1 Introduction

Path-finding is used in many fields, namely network routing tables construction, robotics path planning [Ark86, LS90], global positioning systems (GPS) [SSP08], computer/video games [YS08, VX09], and so forth. In most of these fields, a pathfinder is required to be complete and optimal. Recall that a *complete* pathfinder is one that finds a solution (a path) if one exists in the search space of connected nodes (a graph). The weight or cost of traversing an edge (the connection between two nodes) of a graph usually refers to the Euclidean distance between locations in a game world associated to the search graph. Also, an *optimal* pathfinder is one that always finds the best/lowest cost/shortest path (if one exists), between a start and goal nodes of a graph.

For example, Dijkstra's [Dij59] algorithm is complete and optimal, but the best-first search [Pea84] is a complete, but not an optimal, pathfinder. Like the best-first search, A* [HNR68] and fringe search [BEHS05] also use an heuristic function, but they are complete and optimal, as long as the employed heuristic is admissible, i.e., the heuristic value must always be lower than the real cost from either graph node to the goal node [BM83]. Note that an heuristic is an estimate of the cost from either node to a goal node, e.g., the the Euclidean distance between either node and the goal node. It is an estimate because there may be no edge directly connecting either two nodes. The real cost refers to the actual cost of going from one node to another in the graph traversing only existing edges.

Similarly to the best-first search, the best neighbor first search (BNFS) proposed in this chapter is complete but not optimal. This is so because, in computer/video games, it suffices for a pathfinder to be complete [BMS04], simply because the path does not

need to be the shortest one, what, in a way, mimics what humans do in real-life. Besides, a NPC (non-player character) that always travels the shortest path is hard —if not impossible— to beat in practice.

5.1.1 Research Question

Interestingly, some authors devised sub-optimal pathfinders that are intended to have lower memory requirements and better time performance than A*. In the present chapter, we develop a novel sub-optimal pathfinder that hopefully has lower memory requirements and better time performance than A*.

Therefore, the main research question (RQ) underlying the research work described in this chapter is as follows:

> *Can a novel sub-optimal pathfinder be devised, which has the same or lower memory requirements, better time performance than other pathfinders, and be more near optimal than other sub-optimal pathfinders (e.g., best-first search)?*

5.1.2 Contributions

In this chapter, we introduce a new path-finding algorithm, called best neighbor first search (BNFS), as well as its follow-up algorithm, called trimmed best neighbor first search pathfinder (TBNFS), that mimics the likely erratic behavior of humans when they walk from a point to another, particularly in labyrinths. Besides, BNFS has the advantage that it commonly outperforms Dijkstra's, A*, and fringe search pathfinders in terms of less memory space consumption and better time performance. Also, the TBNFS pathfinder finds better sub-optimal paths than best-first search —the fastest canonical pathfinder— and in some cases, such as BNFS, outperforms best-first search in terms of better time performance.

5.2 BNFS Pathfinder

5.2.1 BNFS Cost Function

The best neighbor first search (BNFS) can be seen as a variant of best-first search (BestFS), because they use the same cost function as follows:

$$F(n) = H(n) \tag{5.1}$$

where $H(n)$ is the cost estimate or heuristic value from the node n to the goal node; our heuristic is the Euclidean distance. The difference is that for BNFS the next node is

the neighbor node with lowest cost estimate, while BestFS does not necessarily select such a neighbor node but the one with lowest cost estimate in the open set; obviously, in the later case, we are assuming that the neighbor nodes of are already in the open set.

However, the choice of the next node n (see lines 20 to 32 of Algorithm 8) must satisfy the following three conditions simultaneously:

1. it is not the parent of current node;
2. it has not been visited;
3. it has the lowest estimate (heuristic) to goal node.

5.2.2 BNFS Algoritm

BNFS behaves as a person with a compass and some chalk trying to reach a location (i.e., the goal or target node) in a map or a maze. When the person reaches a crossing he/she evaluates all the possible directions, and chooses the one with the lowest estimate to reach the target location. The crossing and its incoming and outgoing directions are marked with chalk. In terms of search graph, this is equivalent to mark the incoming (the previous current), crossing (the current), and outgoing (the next current) nodes as visited.

Before proceeding any further, let us make clear important concepts in the context of our BNFS pathfinder as follows:

- The *current node* is the node that is being evaluated, and this means that computing the cost estimates of its neighbor nodes to the goal node is taking place.
- A *visited node* is any node that already was the current node at some moment in the past.
- An *open node* is a visited node that has at least one non-visited neighbor node.
- A *closed node* is a visited node whose eight neighbor nodes are all visited nodes.
- A *hash node* is an open or closed node.

In addition to the search graph, BNFS uses three sets of nodes: (i) the open set $\mathbb{O}$ or set of open nodes; (ii) the closed set $\mathbb{C}$ or set of closed nodes; and (iii) the hash set $\mathbb{H} = \mathbb{O} \cup \mathbb{C}$ or the set of open and closed nodes (i.e., the set of all visited nodes). That is, $\mathbb{C}$ only stores visited nodes whose eight neighbors have been also visited, $\mathbb{H}$ is here used for efficiency sake to quickly check if all the neighbors of a node have been visited, while $\mathbb{O}$ plays the central role in BNFS because it is used for backtracking, particularly when one enters into an alley or a room with a single door. In fact, the last node stored in $\mathbb{O}$ is the last crossing taken during path search.

Algorithm 8 BNFS Algorithm

Input: s: start node
Input: g: goal node
Output: $\mathbb{P}$: path vector holding predecessors of current nodes

1: $\mathbb{O} \leftarrow \emptyset$ {empty open set}
2: $\mathbb{C} \leftarrow \emptyset$ {empty closed set}
3: $\mathbb{H} \leftarrow \emptyset$ {empty hash set}
4: $c \leftarrow s$ {current node is the start node}
5: $\mathbb{O} \leftarrow \mathbb{O} \cup \{c\}$
6: **while** $\mathbb{O} \neq \emptyset$ **do**
7: $\mathbb{H} \leftarrow \mathbb{H} \cup \{c\}$ {add current node to $\mathbb{H}$}
8: **if** $c = g$ **then**
9: $\mathbb{P} \leftarrow$ reconstructPath($\mathbb{P}$, g, s) {path found!}
10: **return** $\mathbb{P}$
11: $N \leftarrow$ neighborNodes(c) {set of neighbors of current node}
12: **if** $N \subset \mathbb{H}$ **then**
13: **if** $c \notin \mathbb{C}$ **then**
14: $\mathbb{C} \leftarrow \mathbb{C} \cup \{c\}$ {all neighbors of c have been visited}
15: **if** $c \in \mathbb{O}$ **then**
16: $\mathbb{O} \leftarrow \mathbb{O} \setminus \{c\}$
17: **if** $\mathbb{O} = \emptyset$ **then**
18: **continue** {No path found!}
19: $c \leftarrow$ lastNode($\mathbb{O}$) {a dead end has been found, so backtracking}
20: **else**
21: $min \leftarrow c$ {when not all neighbors of c have been visited}
22: $cost := \infty$
23: **for** $n \in N$ **do**
24: **if** $n \notin \mathbb{H}$ **and** $n \neq$ parent(c) **then**
25: cost(n) $\leftarrow$ H(n) {heuristic estimate from n to g, Eq. (5.1)}
26: **if** $cost >$ cost(n) **then**
27: $cost \leftarrow$ cost(n)
28: $min \leftarrow n$
29: **if** $c \notin \mathbb{O}$ **then**
30: $\mathbb{O} \leftarrow \mathbb{O} \cup \{c\}$
31: $\mathbb{P}[min] = c$
32: $c = min$
33: **return** null {No path found!}

Note that there is no need to sort $\mathbb{O}$ to find the lowest cost node because this node is chosen among the non-visited nodes neighboring the current node. In fact, the next node is not picked up from the open set $\mathbb{O}$, but there is an exception, which occurs when a dead end is found, that is, when the current node has no more non-visited nodes to go. In this particular case, we have to pick up the last node stored in $\mathbb{O}$ to backtrack to the last crossing node with at least one non-visited node. This means that the BNFS stops when one the following conditions is satisfied: (i) the goal node has been reached; (ii) the open set $\mathbb{O}$ has been emptied.

So, the core of the BNFS can be outlined as follows:

- Select the lowest cost node neighboring the current node (see lines 20 to 32 of Algorithm 8) .
- If a dead end is found before reaching the goal node, backtrack to the last crossing node in $\mathbb{O}$ (see line 19 of Algorithm 8).
- Stop if the goal node is found or $\mathbb{O}$ is emptied (see lines 8-10 and 17-18 of Algorithm 8).

Note that Algorithm 8 considers the following auxiliary functions, namely:

- *lastNode*. This function returns the last node of the open set O.
- *neighborNodes*. This function returns the set N of nodes neighboring the current node c.
- *H*. This is the heuristic function that returns the Euclidean distance from a neighbor node to the goal node.
- *parent*. This function returns the parent of a given node. Note the parenthood of a node is stored in the node itself.
- *reconstructPath*. Reached the goal node, this function reconstructs the path backwards to the start node using the child-parent relationships held in each node.

5.2.3 TBNFS Algoritm

The trimmed BNFS (ou TBNFS) is essentially identical to BNFS. The only difference lies in the way the found path is reconstructed (see line 9 of Algorithm 8). In fact, the path returned in $\mathbb{P}$ may contain loops, i.e., sub-paths whose beginning and ending nodes coincide. As far as TBNFS is concerned, such loops are removed from $\mathbb{P}$.

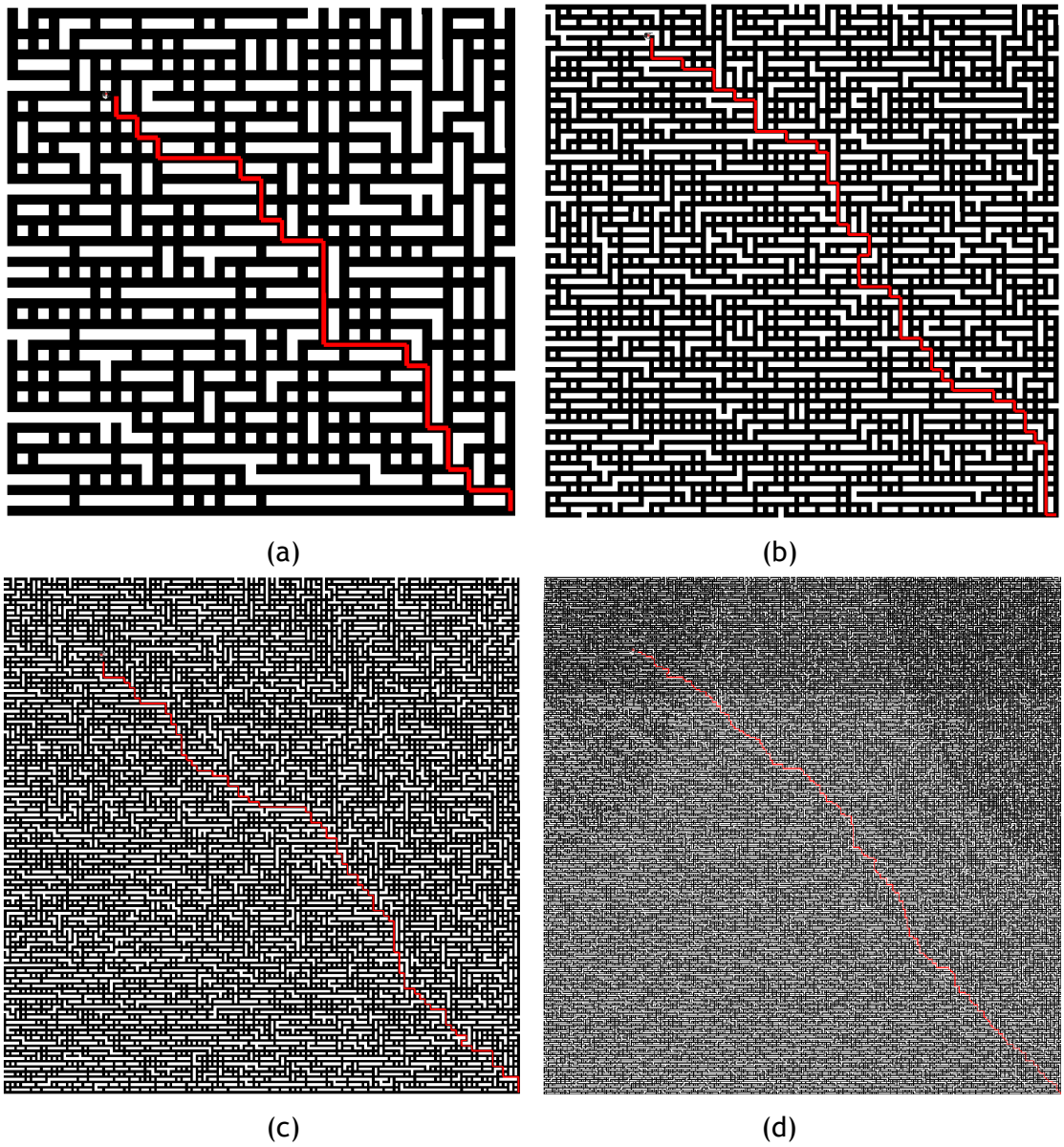


Figure 5.1: Mazes generated using Prim's algorithm: (a) 25x25; (b) 50x50; (c) 100x100; (d) 250x250. Examples of paths are depicted in red.

5.3 Experimental Results

Our testing involved a comparative analysis between Dijkstra (point-to-point variant), A*, best-first search (BestFS), fringe search (FS), BNFS, and TBNFS. We used a set of mazes and HOG2 maps (*Dragon Age: Origins* and *Warcraft 3*) as our ground-truth datasets.

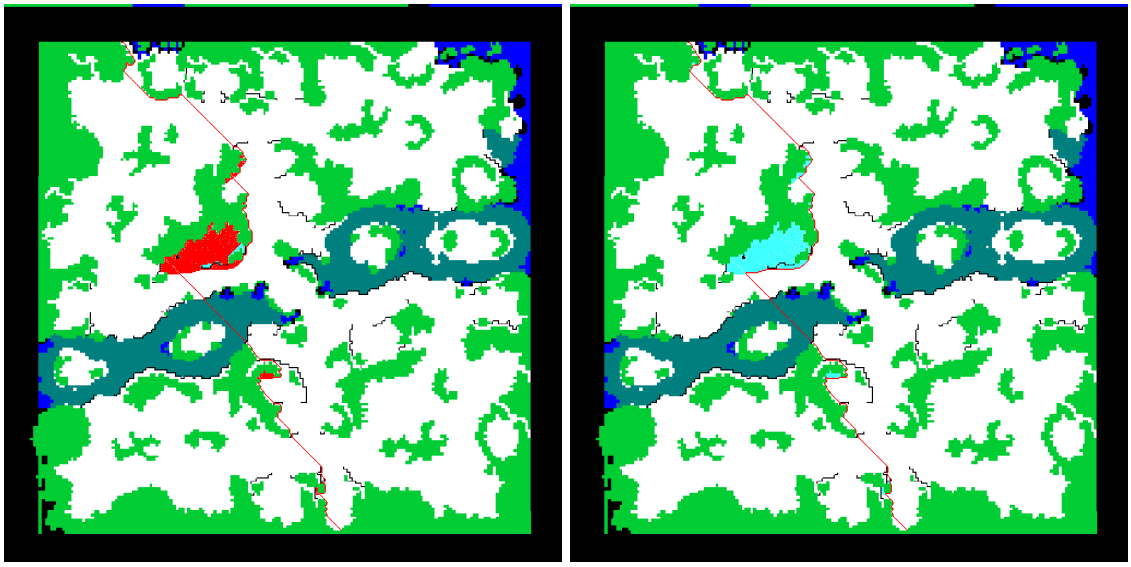
5.3.1 Software/Hardware Setup

The benchmark pathfinders were all implemented in Java programming language. Also, as mentioned in the previous chapters, we used a desktop computer running a Windows

7 64-bit Professional operating system, equipped with an Intel Core i7 920 @ 2.67GHz processor processor, 8 GB Triple-Channel DDR3 de RAM, and a NVIDIA GeForce GTX 295 graphics card with 896MB GDDR3 RAM.

5.3.2 Grid-Based Maps

All pathfinders benchmarked in this paper build upon grid-based maps. We used three types of grid-based maps: (i) maze maps; (ii) indoor maps; (iii) outdoor maps. The maze maps were generated by an in-house maze generator based on Prim's algorithm; examples of maze maps are shown in Fig. 5.1. The indoor and outdoor maps were taken from the HOG2 repository available at `http://movingai.com/benchmarks/`. More specifically, the indoor maps concern the role playing game (RPG) entitled *Dragon Age: Origins*, while the outdoor maps concern the real time strategy (RTS) game entitled *Warcraft 3* (see Fig. 5.2).



(a) (b)

Figure 5.2: HOG2 Map From Warcraft 3: (a) divide_and_conquer Path Found Resorting to BNFS; (b) divide_and_conquer Path Found Resorting to TBNFS. Examples of paths are depicted in red. Visited nodes appear in cyan.

Each HOG2 map file is an ASCII file encoding a grid-based map, whose cells (or squares) encoded as ASCII characters as follows: '.' (passable cell); 'G' (passable cell); '@' (out of bounds); 'O' (out of bounds); 'T' (trees, impassable); 'S' (swamp, passable from regular terrain); and 'W' (water, traversable, but not passable from terrain).

In addition to such ASCII encoding, we have also use a map color encoding for graphics rendering purposes. Each cell color depends on the type of terrain; more specifically, passable is white, out of bounds is black, tree is green, swamp is a greenish blue, and water is blue. Found paths are in red, while visited nodes representing the search space of a given pathfinder are in cyan.

5.3.3 Testing Methodology

In our tests, we adopted the following methodology. First, we set up a dataset for each type of map: mazes, indoor maps (*Dragon Age: Origins*), and outdoor maps (*Warcraft 3*). The mazes used in our tests were generated using Prim's algorithm. More specifically, we generated equally-sized 10 mazes per each one of the following four resolution-based categories: (i) 25×25; (ii) 50×50; (iii) 100×100; and (iv) 250×250. We also used 10 indoor maps and 10 outdoor maps.

Second, we set up the number of paths to be determined per maze or map. One path was generated per maze, in particular a path from the leftmost upper traversable node to the rightmost lowest traversable node. In the case of maps, we generated the following three paths per map: (i) a path from the leftmost upper traversable node to the rightmost lowest traversable node; (ii) a second path between two randomly chosen nodes; (iii) a third path between two randomly chosen nodes. These three paths cannot have identical start nodes nor identical goal nodes.

Third, we measured the optimality of each path i relative to the shortest path found by Dijkstra's algorithm, which can be defined as follows:

$$o = 1 - \frac{p_i - d_i}{d_i} \tag{5.2}$$

where, p_i (number of nodes) refers to the i-th encountered path by a specific pathfinder and d_i (number of nodes) is the optimal Dijkstra's path considering that both paths share the same start and goal nodes.

5.3.4 Mazes

Figs. 5.3 through 5.5 provide the results obtained from testing the six pathfinders mentioned above, including BNFS and TBNFS, in terms of memory consumption, time performance, and path optimality. Testing mazes did not consider diagonal neighbor cells in pathfinding. That is, the 4-neighborhood for each cell only comprises axis-aligned neighbor cells.

Memory consumption. As shown in Fig. 5.3, Dijkstra pathfinder consumes more memory space than any other pathfinder. A* improves on Dijkstra because the heuristic leads to a less expansion of the search space. Recall that both Dijkstra and A* build upon breadth-first searches, that is, their search space expansion is in width.

The remaining four pathfinders spend about half memory space relative to Dijkstra, but fringe search has a slight advantage over the other three pathfinders. This is consistent with the fact that 3 out of 4 lowest memory consumption pathfinders perform depth-first search, namely fringe search, BNFS, and TBNFS, while the best-first search actually performs a pseudo depth-first search because it combines a breadth-first search with a look-ahead heuristic as cost metric.

Time performance. Typically, the less memory-consuming pathfinders are also the quickest because they deal with a less number of nodes. As shown in Fig. 5.4, this rule also applies to mazes, that is, the fringe search, best-first search, BNFS, and TBNFS perform quicker than Dijkstra and A* pathfinders. But, BNFS ranks immediately after the quickest pathfinder, the best-first search.

Path optimality. As expected, as shown in Fig. 5.5, Dijkstra, A* and fringe search are optimal pathfinders, i.e., their path optimality is always 1. The path optimality of the remaining pathfinders is always over 0.9, with best-first search ranking fourth, TBNFS fifth, and BNFS on the last place. This shows that BNFS and TBNFS are suited for maze-based games like PacMan, because they are not optimal, that is, they give a chance to a player win the game.

5.3.5 HOG2 Indoor Maps

Let us now analyze the benchmark results in Figs. 5.6 through 5.8 concerning ten HOG2 indoor maps of the role playing game (RPG) entitled *Dragon Age: Origins* (or DAO shortly).

Memory consumption. As shown in Fig. 5.6, DAO memory consumption follows the same pattern as mazes's one (see Section 5.3.4). Fringe search is always the less memory-consuming pathfinder. In fact, unlike other pathfinders, fringe search does not use open and closed lists at all. Anyway, fringe search, best-first search, BNFS, and TBNFS spend about the same memory space; specifically, they consume one-third less memory space on average than Dijkstra pathfinder. Note that BNFS and TBNFS are slightly more memory-consuming than best-first search because more often enter into alleys than best-first search.

Time performance. As shown in Fig. 5.7, the best-first search is the fastest pathfinder for indoor maps. Nevertheless, the time performance of BNFS (and TBNFS) is comparable to the one of best-first search. The slowest pathfinders are Dijkstra and fringe search, but A* is often slower than fringe search. Note that BNFS outperforms the best-first search for the following maps: brc204d, den500d, and lak304d.

Path optimality. As expected, and shown in Fig. 5.8, Dijkstra, A* and fringe search remain optimal pathfinders for DAO; that is, their path optimality is equal to 1. In regards to best-first search, its path optimality is 0.782 on average. Interestingly, the path optimality of BNFS and TBNS is 0.314 and 0.931 on average, respectively. The fact the the value of o is low for BNFS and high for TBNFS means that a number of path loops were trimmed from the paths returned by BNFS. Therefore, TBNFS surpasses the optimality of best-first search.

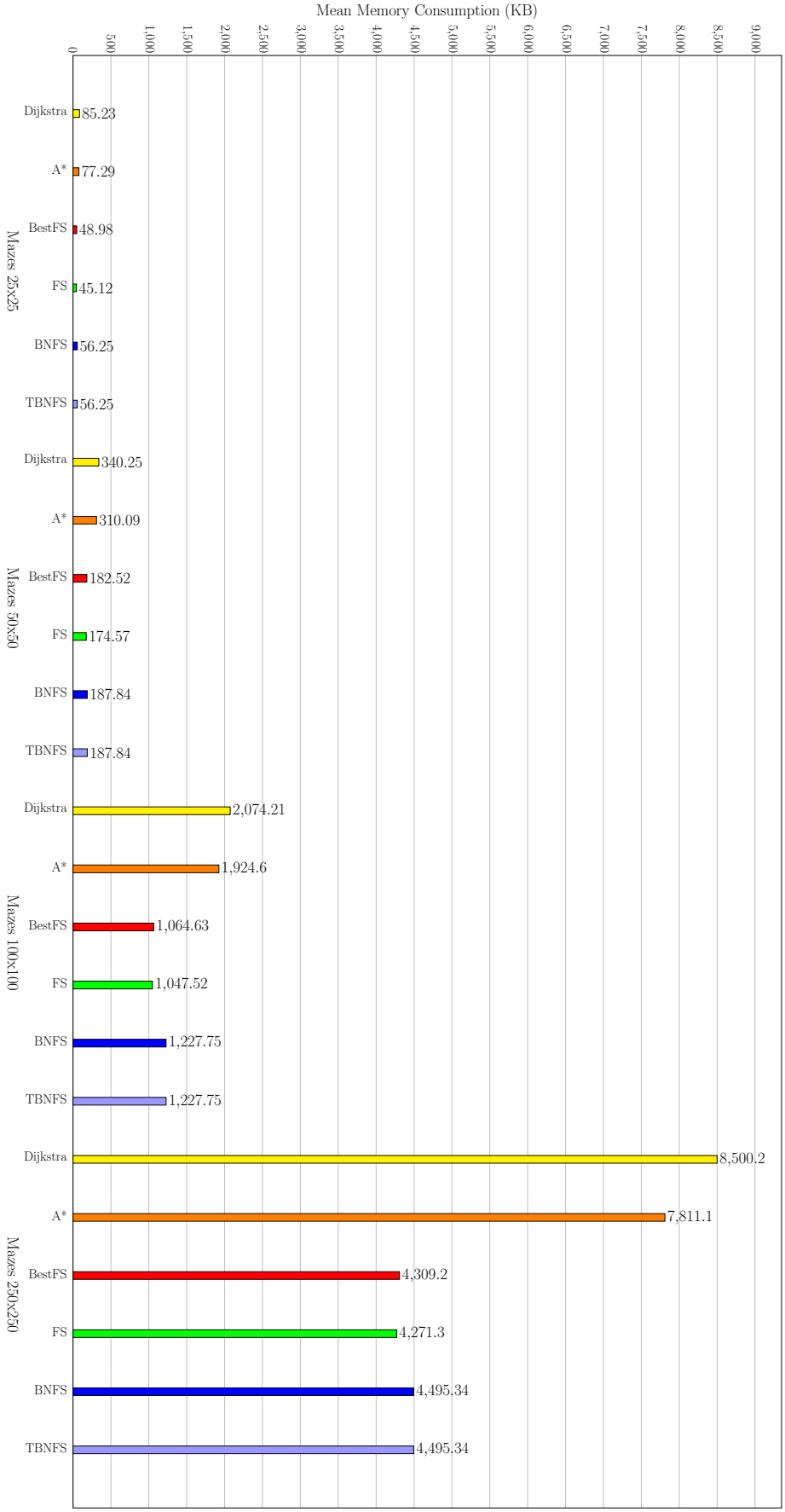


Figure 5.3: Mazes: mean memory consumption results.

Figure 5.4: Mazes: mean time performance results.

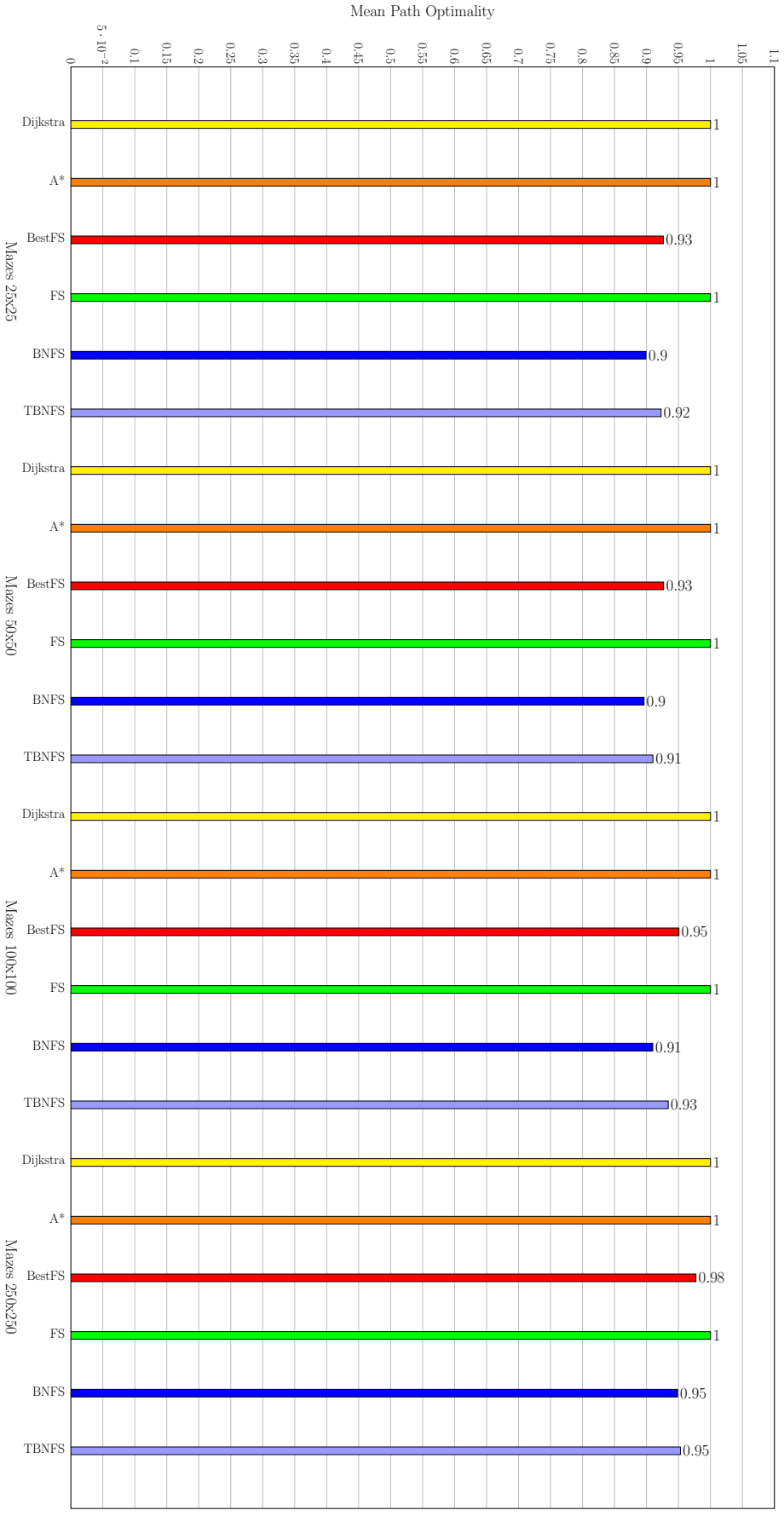


Figure 5.5: Mazes: mean path optimality results.

Figure 5.6: DAO: memory consumption results.

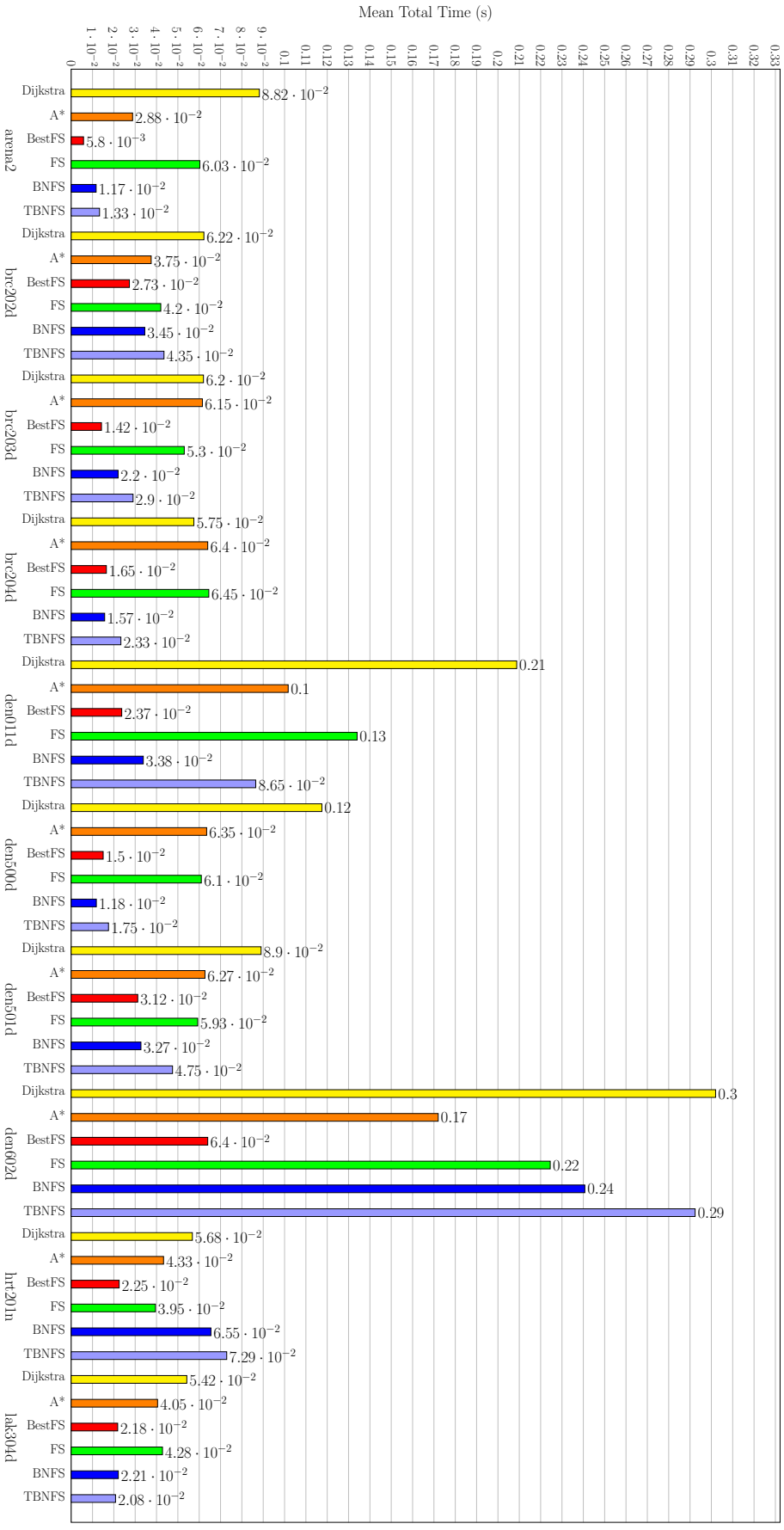


Figure 5.7: DAO: time performance results.

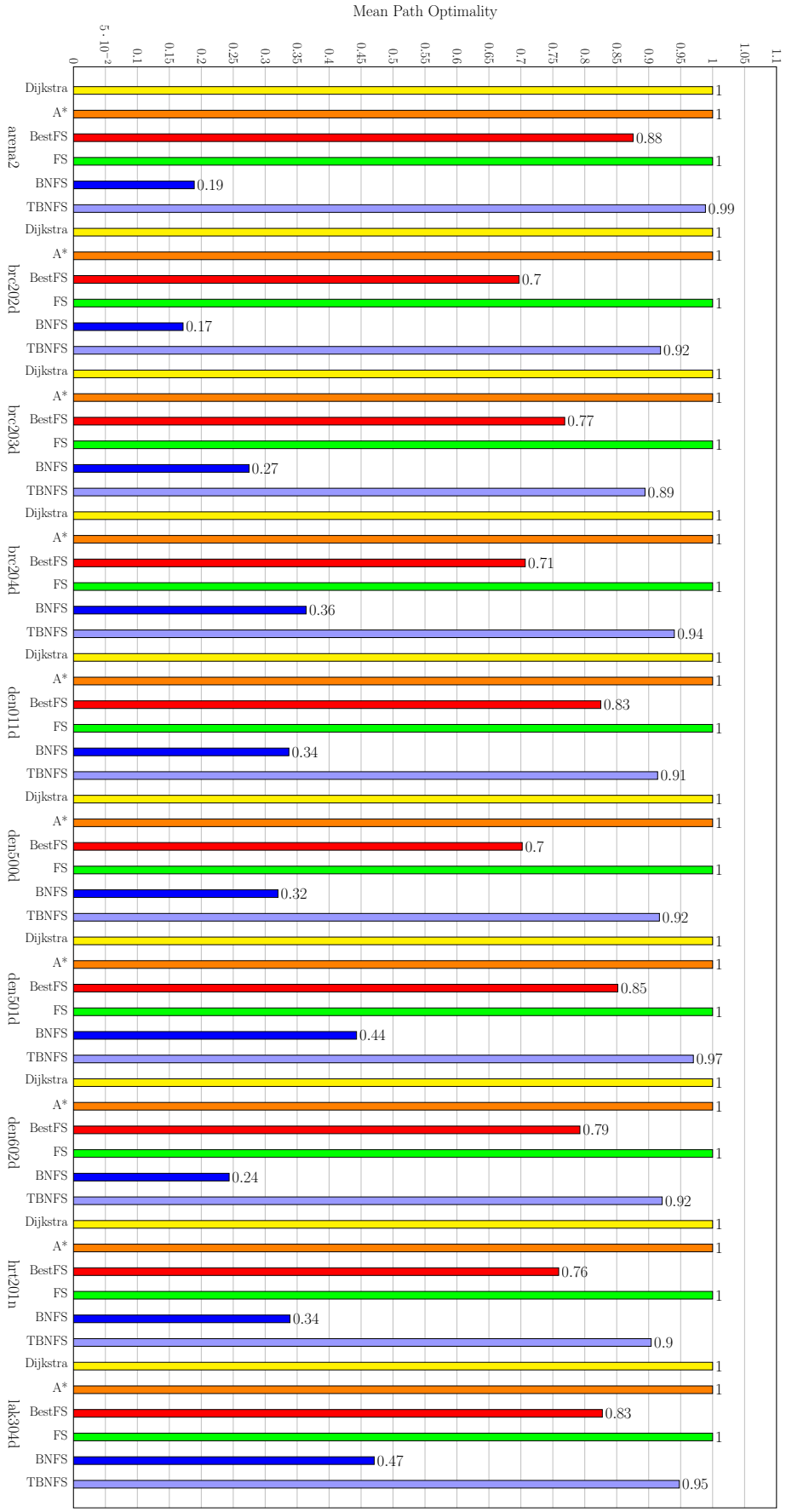


Figure 5.8: DAO: path optimality results.

5.3.6 HOG2 Outdoor Maps

Figs. 5.9-5.11 show the results obtained for ten HOG2 outdoor maps of the real-time strategy (RTS) game entitled *Warcraft 3* (or W3 shortly).

Memory consumption. As shown in Fig. 5.9, W3 memory consumption practically is the same for fringe search, best-first search, BNFS, and TBNFS. These four pathfinders outperform Dijkstra and A* in terms of memory space consumption. In fact, they spend two-fifths less memory space on average than Dijkstra pathfinder, and one-tenth less memory space than A*.

Time performance. As shown in Fig. 5.10, Dijkstra's and fringe search are always the slowest pathfinders for outdoor maps of W3, being seconded A*. The best-first search is the fastest pathfinder for outdoor maps, while BNFS and TBNFS rank second and third, respectively, though the latter outperform the best-first search in the HOG2 harvestmoon map.

Path optimality. As expected, Fig. 5.11 confirms that Dijkstra, A* and fringe search are optimal pathfinders for W3 ($o = 1$). The path optimality of the best-first search is 0.841 on average. Interestingly, the path optimality of BNFS and TBNS is 0.571 and 0.911 on average, respectively; these values are similar to the ones obtained for DAO. The high path optimality value of TBNFS is a result of trimming path loops of the original BNFS paths. Note that again TBNFS surpasses the optimality of best-first search.

5.3.7 Further Discussion

Let us now discuss benchmarking pathfinders (including BNFS and TBNFS) in more general terms. As illustrated in Figs. 5.3, 5.6, and 5.9, fringe search, best-first search, BNFS, and TBNFS present comparable results regarding *memory space consumption*, and rank ahead Dijkstra and A*, no matter the type of the input map, be it a maze, a DAO or a W3 map.

In terms of time performance, and taking into account the results presented in Figs. 5.4, 5.7, and 5.10, BNFS outperforms Dijkstra, A*, fringe search, and in some cases the best-first search.

Finally, and considering the results presented in Figs. 5.5, 5.8, and 5.11, TBNFS enjoys the best path optimality among the sub-optimal pathfinders, including the best-first search and BNFS. In fact, TBNFS produces paths that are very close to optimal paths, because path loops are trimmed. In true, TBNFS is a path loop pruning technique that applies to any pathfinder.

Figure 5.9: W3: memory space consumption results.

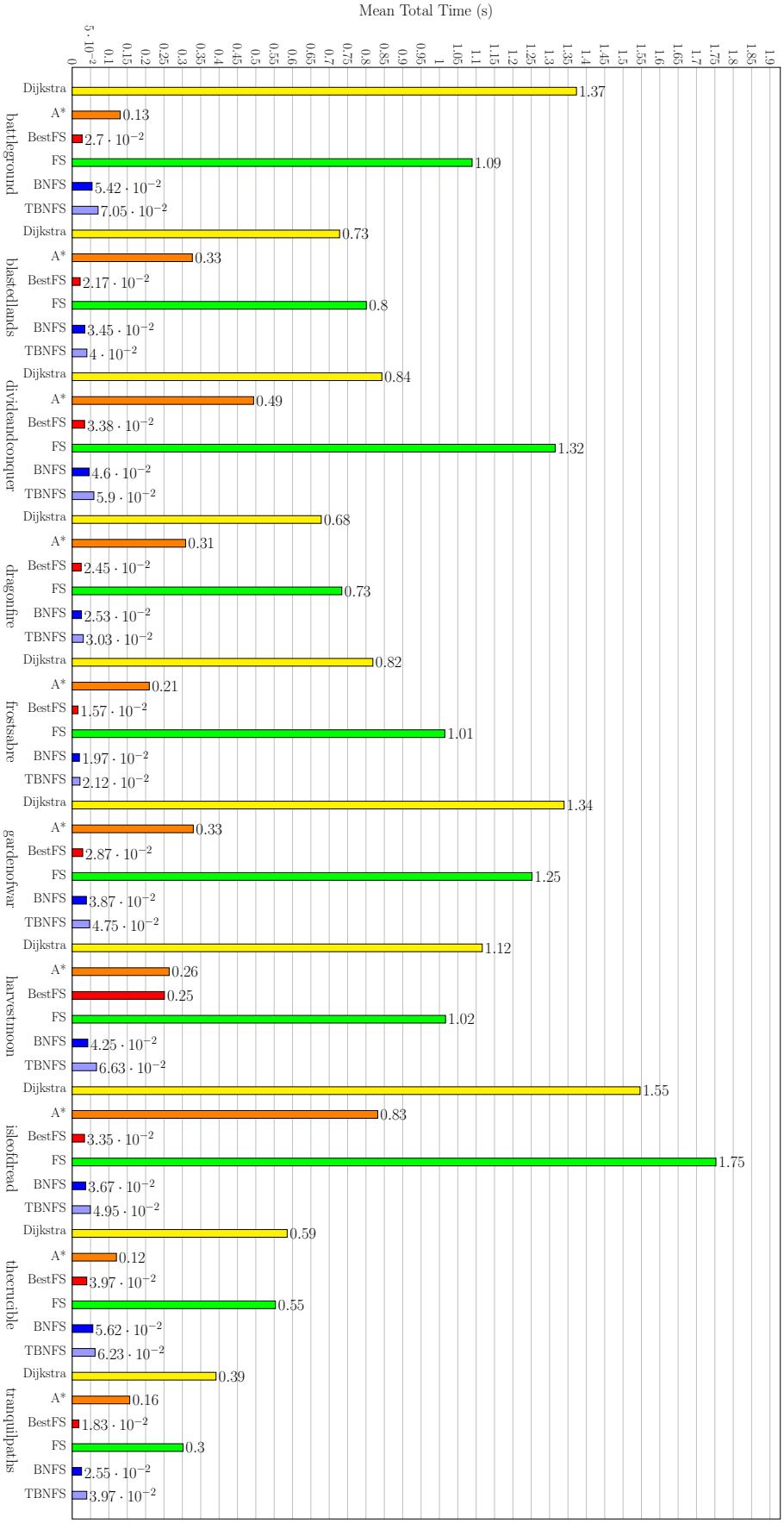


Figure 5.10: W3: time performance results.

Figure 5.11: W3: path optimality results.

5.4 Further Remarks

We have introduced the best neighboor first search (BNFS) and its follow-up TBNFS. Specifically, they were compared to Dijkstra, A*, fringe, and best-first searches. In general, BNFS achieves better performance than those pathfinders either in terms of less memory space consumption or in terms of time. So, we can respond affirmatively to the research question put forward in the beginning of the current chapter. We have also introduced a path loop pruning technique that was applied to BNFS, resulting in a loop-free pathfinder called trimmed best neighbor first search (TBNFS). However, such a pruning technique can be applied to any other pathfinder.

Chapter 6

Conclusions

This thesis describes a few contributions to the advance of knowledge in the video games field, mostly in respect to the integration of pathfinders and influence fields. It is also shown that not all pathfinders can be merged with influence fields, at least in the manner described in this thesis.

6.1 Research Challenges

The research work behind this thesis has been developed on the basis of the following challenges:

1. There was not an universal technique that integrated influence fields with canonical pathfinders (e.g. A*, Dijkstra, best-first search). Note that a first merging pathfinding and influence maps was attempted by Paanakker [Paa08] in an algorithm called risk-adverse pathfinding. However, it did not work well for attractors and required a costumized setup per map.

2. There was no work that integrated fields with pathfinders to simultaneously solve the problems of path adaptivity, path smoothness, reduction of memory space consumption, and increase of time performance. In this thesis, the path smoothness issue was not addressed deeply.

The use of influence fields combined with the current state-of-the-art pathfinders has proven to be a successful approach, with significant gains in less memory and time consumption. One of the novelties of our approach is that adding, moving or removing propagators solely requires updating the influence field locally in the surroundings of each propagator. This allows us to easily update (if required) the influence field before each iteration of the pathfinder, without the need to recalculate the path from the start node again.

Remarkably, in either of the our novel techniques, the use of influence fields does not create local extremum issues when combined with pathfinders. This is so because the nature of a pathfinder remains the same, i.e., the agent always moves forward to the next node on the way to goal node. That is, even when the agent is moving towards an attractor (i.e., a possible local minimum), it does not stop walking when it arrives at the propagator, because the pathfinder always determines the next node to go.

6.2 Thesis Achievements

Regarding our thesis statement, it is safe to state that influence fields indeed can be integrated with pathfinders to address their shortcomings (e.g. local extremum problem and path adaptivity). Specifically, two adaptive pathfinding algorithms (see Chapters 3 and 4) that merge A*, Dijkstra's algorithm, and best-first search with influence fields were proposed. As an extra contribution, a novel sub-optimal pathfinder, called best neighbor first search (BNFS and its follow-up trimmed BNFS), was introduced as a result of unsuccessfully integrating fringe search with influence fields. More importantly, this thesis has opened a window for a novel family of real-time adaptive pathfinders suitable to video games.

6.3 Research Limitations

Our pathfinding-and-influence approach does not work with depth-first search algorithms (e.g., IDA*, fringe search, BNFS, and TBNFS). Specifically, the proposed techniques only work when the open set nodes are accessed in lowest cost order. Another limitation resides in its inability to handle influence field changes when a node within the open set changes its cost for any reason. That is to say that, if an already evaluated but not closed node changes its cost, as a consequence of a propagator removal/addition in the influence field, there are no guarantees that in the vicinity of such node a repeller will be avoided or a attractor will force the path through it.

6.4 Future Work

Regarding influence field adaptive A* algorithms, future work includes:

1. *Identify more pathfinders that, like A* and its variants, can be altered with either of the proposed pathfinding-and-influence techniques*. Specifically, we are interested in to know whether or not our pathfinding-and-influence techniques also apply to general-purpose sub-optimal bounded pathfinders, as well to lazy theta* [NKT10] and jump-point search [HG11]. For that purpose, we need to implement and benchmark them against the ground-truth HOG2 maps. The benchmarking pathfinders must include the risk-adverse pathfinding [Paa08] and constraint-aware navigation [KNS+13]. The idea of this point is to access how close to universal are our two pathfinding-and-influence techniques.

2. *Design and implementation of hierarchical influence-aware pathfinders*. This may open the window to engender multi-agent path planning using pathfinding-and-influence techniques. The idea of a hierarchical influence-aware pathfinder is to

allow for its parallelization. Designing a hierarchical pathfinder requires to partition the search graph into zones, so that paths are determined from possible entry and exit points of zone in parallel. Similarly, to update the influence field in parallel requires its partition into regions, updating then any region that has been subject to the addition or removal of a repeller/attractor.

3. *Design and implementation of complex simulation scenarios where multiple agents perform path-finding in a dynamically changing influence field with moving propagators*. In thesis, we have only considered static propagators. The idea here is to design and implement complex simulation scenarios where each moving agent or NPC is associated with a influence field. Therefore, our intent is not only to implement effective ways to store each NPC's influence field with as least memory as possible, but also to study how our pathfinding-and-influence techniques are relevant for multi-agent pathfinding.

4. *Compare our pathfinding-and-influence techniques with those underlying adaptive pathfinders such as, for example, D* or any-angle path planning algorithms (e.g., theta** [NDKF07]*)*. The big picture is to assess how adaptivity affects not only our two influence-aware techniques, but also other adaptive pathfinders, namely lifelong planing A* (LPA*) [KLF04] and D* family [Ste94, Ste95, KL05] of A* variants, particularly where NPCs and other moving agents carry influence fields, thats is, when they work as moving propagators. Besides, we are interested in knowing how these dynamic propagators interact with static propagators as those used in this thesis. Obviously, the automated placement of static propagators in the game world is another open issue to be investigated in the future.

6.5 Concluding Remarks

Essentially, this thesis has shown that it is feasible to integrate pathfinding with influence fields, resulting in adaptive pathfinders that are sensitive to changes in the game world, not matter whether such changes are static (e.g., destruction of a bridge) or dynamic (e.g., a moving NPC carrying a propagator).

Bibliography

[AAG15] Michael Adaixo, Gonçalo. Amador, and Abel Gomes. Algoritmo de Dijkstra com Mapa de Influência de Atratores e Repulsores (*in Portuguese*). In *Proceedings of the 2015 Portuguese Conference of Science and Art in Video Games* (SciTecIN'15), Coimbra, Portugal, Nov. 12 - 13, 2015. xxiii, 6, 83, 84

[All11] Thomas L. P. Allen. *Time-Optimal Active Decision Making*. PhD thesis, ARC Centre of Excellence in Autonomous Systems, Australian Centre for Field Robotics, School of Aerospace, Mechanical and Mechatronic Engineering, The University of Sydney, 2011. 82

[Ang12] Bobby Anguelov. Video game pathfinding and improvements to discrete search on grid-based maps. Master's thesis, University of Pretoria, 2012. 15

[Apo69] T. M. Apostol. *Calculus. Volume II: Multi-Variable Calculus and Linear Algebra, with Applications to Differential Equations and Probability*. John Wiley & Sons, Inc., New York, 2nd edition edition, 1969. 33

[APS08] Leigh Achterbosch, Robyn Pierce, and Gregory Simmons. Massively Multiplayer Online Role-Playing Games: The Past, Present, And Future. *Comput. Entertain.*, 5(4):1–33, March 2008. 11

[Ark86] Ronald C. Arkin. Path Planning for a Vision-Based Autonomous Robot. Technical report, Amherst, MA, USA, 1986. xviii, 1, 26, 33, 64, 85

[Ark87] Ronald Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation* (ICRA'87), Raleigh, NC, USA, March, pages 264–271. IEEE Computer Society Press, 1987. 26, 33, 64

[ASK15] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *Int. J. Comput. Games Technol.*, January 2015. Available from: `http://dx.doi.org/10.1155/2015/736138`. xix, 2, 64, 82

[Ass15] Information Resources Management Association. *Gamification: Concepts, Methodologies, Tools, and Applications*. IGI Global, Hershey, PA, USA, 1st edition, 2015. 11

[BB12] Sandy Brand and Rafael Bidarra. Multi-core pathfinding with Parallel Ripple Search. *Comput. Animat. Virtual Worlds*, 23(2):73–85, March 2012. xviii, 2

[BEHS05] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer. Fringe Search: Beating A* at Pathfinding on Game Maps. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games*

(CIG'05), Colchester, UK, April 4 - 6, pages 125–132. IEEE Computer Society Press, 2005. xviii, 2, 13, 21, 32, 64, 85

[BM83] A. Bagchi and A. Mahanti. Search Algorithms Under Different Kinds of Heuristics-A Comparative Study. *J. ACM*, 30(1):1–21, January 1983. 17, 85

[BMS04] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004. xix, xxii, 2, 5, 32, 37, 64, 85

[BS04] David M Bourg and Glenn Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 1 edition, 2004. xvii, 1, 11

[dB78] Carl de Boor. *A Practical Guide to Splines*. Springer-Verlag, New York, USA, 1978. xxii, 5

[DeL01] Mark DeLoura. *Game Programming Gems 2*. Charles River Media, Inc., Rockland, MA, USA, 2001. 26, 33

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959. xvii, 1, 11, 12, 13, 31, 38, 64, 83, 85

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, 32(3):505–536, July 1985. 18

[DSC08] LeMinh Duc, Amandeep Singh Sidhu, and Narendra S. Chaudhari. Hierarchical Pathfinding and AI-based Learning Approach in Strategy Game Design. *Int. J. Comput. Games Technol.*, 2008:1–11, January 2008. xviii, 2

[FC80] F. N. Fritsch and R. E. Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17:238–246, 1980. xxiii, 5

[Ger06] Johnson Geraint. Smoothing a Navigation Mesh Path. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, pages 129–140. Charles River Media, Inc., 2006. xix, 2, 64

[GMN94] Subrata Ghosh, Ambuj Mahanti, and Dana S. Nau. ITS: an efficient limited-memory heuristic tree search algorithm. In *Proceedings of the 12th National Conference on Artificial Intelligence* (AAAI'94) Seattle, Washington, USA, Aug. 1 - 4, pages 1353–1358. AAAI Press, 1994. xviii, 2, 21, 32

[Goo00] M.A. Goodrich. Potential Fields Tutorial. Internet, 2000. Available from: `http://borg.cc.gatech.edu/ipr/files/goodrich_potential_fields.pdf`. xx, 3, 27, 65

[HBUK12] Carlos Hernández, Jorge Baier, Tansel Uras, and Sven Koenig. Time-bounded adaptive A*. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems* (AAMAS'12), Valencia, Spain, June 4 - 8, pages 997–1006. International Foundation for Autonomous Agents and Multiagent Systems, 2012. xviii, 2

[HG11] Daniel Damir Harabor and Alban Grastien. Online Graph Pruning for Pathfinding On Grid Maps. In *Proceedings of the 25th National Conference on Artificial Intelligence* (AAAI'11) San Francisco, CA, USA, Aug. 7 - 11, pages 1114–1119. AAAI Press, 2011. 106

[HJ08a] Johan Hagelbäck and Stefan Johansson. The Rise of Potential Fields in Real Time Strategy Bots. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference* (AIIDE'08), Stanford, CA, USA, Oct. 22 - 24, pages 42–47. AAAI Press, 2008. 26, 33

[HJ08b] Johan Hagelbäck and Stefan Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS'08), Estoril, Portugal, May 12 - 16, pages 631–638. International Foundation for Autonomous Agents and Multiagent Systems, 2008. xix, 3, 26, 33, 34, 64

[HJ09a] Johan Hagelbäck and Stefan J. Johansson. A Multi-Agent Potential Field-based Bot for a Full RTS Game Scenario. In *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference* (AIIDE'09), Stanford, CA, USA, Oct. 14 - 16, pages 28–33. AAAI Press, 2009. 26, 33

[HJ09b] Johan Hagelbäck and Stefan J. Johansson. A Multiagent Potential Field-based Bot for Real-time Strategy Games. *Int. J. Comput. Games Technol.*, 2009:1–10, January 2009. 26, 33

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. xvii, 1, 12, 13, 31, 64, 83, 85

[Kha85] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation* (ICRA'85) St. Louis, MO, USA, March 25 - 28, pages 500–505. IEEE Computer Society Press, 1985. 26, 33, 64

[KK16] Marcelo Kallmann and Mubbasir Kapadia. Geometric and Discrete Path Planning for Interactive Virtual Worlds. In *Proceedings of the 43rd Annual Conference on Computer Graphics and Interactive Techniques: Course Notes: Course Notes* (SIGGRAPH'16), Anaheim, CA, USA, July 24 - 28, pages 1–29. ACM Press, 2016. 28

[KL05] S. Koenig and M. Likhachev. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics*, 21(3):354–363, 2005. xviii, 2, 13, 32, 63, 64, 107

[KLF04] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong Planning A*. *Artif. Intell.*, 155(1-2):93–146, May 2004. xviii, 2, 13, 32, 63, 64, 107

[KNS+13] Mubbasir Kapadia, Kai Ninomiya, Alexander Shoulson, Francisco Garcia, and Norman Badler. Constraint-Aware Navigation in Dynamic Environments. In *Proceedings of the 6th International Conference on Motion in Games* (MIG'13), Dublin, Ireland, Nov. 7 - 9, pages 111–120. ACM Press, 2013. 28, 106

[Kor85] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, September 1985. xviii, 2, 13, 21, 32, 64

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. xvii, 1, 11

[LCCFL13] R. Lara-Cabrera, C. Cotta, and AJ. Fernandez-Leiva. A review of computational intelligence in RTS games. In *Proceedings of the 2013 IEEE Symposium on Foundations of Computational Intelligence* (FOCI'13), Singapore, Singapore, April 16 - 19, pages 114–121. IEEE Computer Society Press, 2013. xx, 3, 26, 33, 34

[LLM07] R. Leigh, S. J. Louis, and C. Miles. Using a Genetic Algorithm to Explore A*-like Pathfinding Algorithms. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games* (CIG'07), Honolulu, HI, USA, April 1 - 5, pages 72–79. IEEE Computer Society Press, 2007. 11

[LM05] S.J. Louis and C. Miles. Playing to learn: case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation*, 9(6):669–681, December 2005. 26, 33

[LMP+13] S. M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius. *Artificial and Computational Intelligence in Games*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik GmbH, 2013. 11

[LR04] Tim Laue and Thomas Röfer. A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields. In Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VII*, pages 122–133. Springer-Verlag, 2004. 27, 33, 63, 65, 83

[LS90] V. J. Lumelsky and A. A. Stepanov. Autonomous robot vehicles. chapter Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape, pages 363–390. Springer-Verlag New York, Inc., New York, NY, USA, 1990. xviii, 1, 85

[Mac96] Annales de l'Académie de Mâcon. 1896. 11

[Mar02] Pinter Marco. Realistic turning between waypoints. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 186–192. Charles River Media, Inc., 2002. xix, 2, 64

[McC07] John McCarthy. What is artificial intelligence? url = http://www-formal.stanford.edu/jmc/whatisai.pdf, 2007. Online; accessed 29 June 2017. xvii, 1

[Mil06] I. Millington. *Artificial Intelligence for Games*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. 27, 65, 66

[MKH99] Vikram Manikonda, P. S. Krishnaprasad, and James Hendler. Mathematical control theory. chapter Languages, Behaviors, Hybrid Architectures, and Motion Control, pages 200–226. Springer-Verlag New York, Inc., 1999. 11

[ML06] C. Miles and S.J. Louis. Towards the Co-Evolution of Influence Map Tree Based Strategy Game Players. In *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games* (CIG'06), Reno, NV, USA, May 22 - 24, pages 75–82. IEEE Computer Society Press, 2006. 26, 33

[Moo57] Edward Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching* April 2 - 5, pages 285–292. Harvard University Press, 1957. 11

[MQLL07] C. Miles, J. Quiroz, R. Leigh, and S.J. Louis. Co-Evolving Influence Map Tree Based Strategy Game Players. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games* (CIG'07), Honolulu, HI, USA, April 1 - 5, pages 88–95. IEEE Computer Society Press, 2007. 26, 33

[Nar04] Alexander Nareyek. AI in Computer Games. *ACM Queue*, 1:58–65, 2004. 11

[NDKF07] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Feiner. Theta*: Any-angle Path Planning on Grids. In *Proceedings of the 22nd National Conference on Artificial Intelligence* (AAAI'07), Vancouver, British Columbia, Canada, July 22 - 26, pages 1177–1183. AAAI Press, 2007. 66, 107

[Nil98] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 31, 63

[NKT10] Alex Nash, Sven Koenig, and Craig Tovey. Lazy Theta*: Any-angle Path Planning and Path Length Analysis in 3D. In *Proceedings of the 24th National Conference on Artificial Intelligence* (AAAI'10), Atlanta, Georgia, USA, July 11 - 15, pages 147–154. AAAI Press, 2010. 106

[OSU+13] S. Ontanon, G. Synnaeve, A Uriarte, F. Richoux, D. Churchill, and M. Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(4):293–311, December 2013. xx, 3, 26, 33, 34

[Paa08] F. Paanakker. *AI Game Programming Wisdom 4*, chapter Risk-Adverse Pathfinding Using Influence Maps, pages 173–178. Charles River Media, Inc., 2008. xvii, 1, 11, 27, 65, 66, 83, 84, 105, 106

[Par92] Parent, A. and Morin, M. and Lavigne, P. Propagation of super-Gaussian field distributions. *Optical and Quantum Electronics*, 24(9):S1071–S1079, 1992. 67

[Pea84] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. 1984. Available from: `http://www.osti.gov/scitech/biblio/5127296`. 85

[PM10] David Poole and Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, Cambridge, United Kingdom, 2010. xvii, 1

[Poh70a] Ira Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236, 1970. xxii, 5, 13, 21, 32, 64

[Poh70b] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193–204, 1970. Available from: `http://www.sciencedirect.com/science/article/pii/000437027090007X`. xvii, 1, 12, 31, 83

[Poh73] Ira Pohl. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (IJCAI'73), Stanford, USA, USA, Aug. 20 - 23, pages 12–17. Morgan Kaufmann Publishers Inc., 1973. xviii, xxii, 2, 5, 13, 21, 32, 64

[Rab00] Steve Rabin. A* Aesthetic Optimizations. In Mark DeLoura, editor, *Game Programming Gems*, pages 264–271. Charles River Media, Inc., 2000. xix, xxii, 2, 5, 32, 64

[Rab02] Steve Rabin, editor. *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA, 2002. xvii, 1, 11, 26

[Rab13] Steven Rabin. *Game AI Pro: Collected Wisdom of Game AI Professionals*. A. K. Peters, Ltd., Natick, MA, USA, 2013. 11

[Rab15] Steven Rabin. *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. A. K. Peters, Ltd., Natick, MA, USA, 2015. 11

[Ree99] B. Reese. AlphA*: an ε-admissible heuristic search algorithm. 1999. Available from: `http://home1.stofanet.dk/breese/papers.html`. xviii, 2, 13, 21, 32, 64

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. xvii, 1, 11

[Run01] Carl Runge. über empirische funktionen und die interpolation zwischen äquidistanten ordinaten. *Zeitschrift für Mathematik und Physik*, 46:224–243, 1901. xxii, 5

[SJ12] J. Svensson and S.J. Johansson. Influence Map-based controllers for Ms. PacMan and the ghosts. In *Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games* (CIG'12), Granada, Spain, Sept. 11 - 14, pages 257–264. IEEE Computer Society Press, 2012. xx, 3, 26, 33, 34

[SKY08] Xiaoxun Sun, Sven Koenig, and William Yeoh. Generalized Adaptive A*. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS'08), Estoril, Portugal, May 12 - 16, pages 469–476. International Foundation for Autonomous Agents and Multiagent Systems, 2008. xviii, 2, 13, 64

[SSP08] B. Sobota, C. Szabo, and J. Perhac. Using path-finding algorithms of graph theory for route-searching in geographical information systems. In *Proceedings of the 6th International Symposium on Intelligent Systems and Informatics* (SISY'08), Subotica, Serbia, Sept. 26 - 27, pages 1–6. IEEE Computer Society Press, 2008. xviii, 1, 85

[Ste94] Anthony Stentz. Optimal and Efficient Path Planning for Partially-Known Environments. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* (ICRA'94), San Diego, CA, USA, May 8 - 13, pages 3310–3317. IEEE Computer Society Press, 1994. xviii, 2, 13, 32, 63, 64, 107

[Ste95] Anthony Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial intelligence* (IJCAI'95), Montreal, Quebec, Canada, Aug. 20 - 25, pages 1652–1659. Morgan Kaufmann Publishers Inc., 1995. xviii, 2, 13, 32, 63, 64, 107

[Stu12] N.R. Sturtevant. Benchmarks for Grid-Based Pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(2):144–148, 2012. xxi, 4, 15, 28

[TBS04] Christian Thurau, Christian Bauckhage, and Gerhard Sagerer. Learning human-like movement behavior for computer games. In *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior* (SAB'04), pages 315–323. MIT Press, 2004. 26, 33, 64

[TPA12] Gill R. Tsouri, Alvaro Prieto, and Nikhil Argade. A Modified Dijkstra's Routing Algorithm for Increasing Network Lifetime in Wireless Body Area Networks. In *Proceedings of the 7th International Conference on Body Area Networks* (BodyNets'12), Oslo, Norway, Sept. 24 - 26, pages 166–169. ICST (Institute for

Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012. xviii, 1

[TR08] Jordan Thayer and Wheeler Ruml. Faster than weighted A*: An optimal approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling* (ICAPS'08), Sydney, Australia, Sept. 14 - 18, pages 355–362. AAAI Press, 2008. xxii, 5, 14, 21, 32, 64

[TR09] Jordan Thayer and Wheeler Ruml. Using Distance Estimates in Heuristic Search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling* (ICAPS'09), Thessaloniki, Greece, Sept. 19 - 23, pages 382–385. AAAI Press, 2009. 21, 32, 64

[TR10] Jordan Thayer and Wheeler Ruml. Finding Acceptable Solutions Faster Using Inadmissible Information. In *Proceedings of the Third Annual Symposium on Combinatorial Search* (SoCS'10), Atlanta, GA, USA, July 8 - 10. AAAI Press, 2010. xviii, xxii, 2, 5, 14, 21, 32, 42, 64, 72

[TS01] Matti Tommiska and Jorma Skyttä. Dijkstra's Shortest Path Routing Algorithm in Reconfigurable Hardware. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (FPL'01), Belfast, Northern Ireland, UK, Aug. 27 - 29, pages 653–657. Springer-Verlag, 2001. xviii, 1

[Var12] Moshe Vardi. Artificial Intelligence: Past and Future. *Communications of the ACM*, 55(1):5–5, January 2012. 11

[VX09] K. Venkataramanan and X. J. Xin. Game Artificial Intelligence Literature Survey on Game AI. http://www.nus.edu.sg/nurop/2009/FoE/U058557W.PDF, 2009. last access at 8 - 12 - 2011. xvii, 1, 85

[WG08] N. Wirth and M. Gallagher. An influence map model for playing Ms. Pac-Man. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games* (CIG'08), Perth, Australia, Dec. 15 - 18, pages 228–233. IEEE Computer Society Press, 2008. 26, 33

[YBHS11] Peter Yap, Neil Burch, Robert Holte, and Jonathan Schaeffer. Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning. In *Proceedings of the 25th National Conference on Artificial Intelligence* (AAAI'11) San Francisco, CA, USA, Aug. 7 - 11. AAAI Press, 2011. 66

[YQS+16] Liang Yang, Juntong Qi, Dalei Song, Jizhong Xiao, Jianda Han, and Yong Xia. Survey of Robot 3D Path Planning Algorithms. *J. Control Sci. Eng.*, 2016. 11

[YS08] Sule Yildirim and Sindre Berg Stene. A survey on the need and use of AI in game agents. In *Proceedings of the 2008 Spring Simulation Multiconference,*

(SpringSim'08), Ottawa, Canada, April 14 - 17, pages 124–131. Society for Computer Simulation International, 2008. xvii, 1, 85

[Zob69] Albert Zobrist. A Model of Visual Organization for the Game of GO. In *Proceedings of the 1969, Spring Joint Computer Conference* (AFIPS'69), Boston, Massachusetts, USA, May 14 - 16, pages 103–112. ACM Press, 1969. 11, 26, 33, 65