Kapitola 10

Algoritmy umělé inteligence

Umělá inteligence představuje součást informatiky, která v průběhu 40 let od svého formálního ustanovení jako vědní disciplína prošla dynamickým vývojem, naplněným jak významnými objevy, tak i vystřízlivěním z přehnaných představ o jejích možnostech. Charakteristická je i skutečnost, že přes významné postavení, které v informatice zaujímá, nepanuje v odborné veřejnosti jediné všeobecně uznávané vymezení obsahu této vědní disciplíny.

Problematika umělé inteligence je natolik rozsáhlá, že předem vylučuje možnost podat v rámci jedné kapitoly její reprezentativní přehled. V souladu s celkovým zaměřením tohoto textu věnujeme pozornost pouze jedinému tématu – klasické problematice řešení úloh jakožto hledání ve stavovém prostoru. Poskytne nám totiž nový pohled na problematiku řešenou v jiném kontextu již dříve (v kap. 3 a 5).

Pro systematické seznámení s umělou inteligencí můžeme vedle nabídky specializovaných předmětů doporučit ještě studium literatury (např. [25], [36], [18], apod.).

10.1 Stavový prostor a řešení úloh

Umělou inteligencí (UI) je možné se zabývat z mnoha hledisek, počínaje obecnými filosofickými otázkami o mentálních možnostech strojů a konče technickým návrhem automatické podmořské nebo kosmické sondy (i když tento příklad spadá již spíše do **robotiky**, která se z UI vyčlenila jako samostatný obor). M. Minsky, jeden ze zakladatelů oboru, charakterizuje UI jako *vědu o vytváření strojů nebo systémů, které budou při řešení určitého úkolu používat takového postupu, který bychom považovali za projev inteligence, pokud by jej používal člověk.*

Stručnější definice zdůrazňuje zakotvení UI ve výpočetní technice: *UI je vědecký obor, který se snaží vysvětlit a napodobit inteligentní chování prostřednictvím výpočetních procesů.* Tato charakterizace zároveň zdůvodňuje i multidisciplinárnost UI, v níž se prolíná informatika s matematikou, inženýrstvím, psychologií, neurologií, atd. Z inženýrského hlediska lze smysl UI vidět ve *vytváření reprezentací a postupů, které umožňují automaticky (autonomně) řešit problémy řešené člověkem.*

Má-li nějaký výpočetní proces napodobit inteligentní lidské chování, je třeba především poznat zákonitosti lidského myšlení. Jak probíhá vytváření konceptů v lidské mysli? Jak se tvoří a používají strategie řešení úloh? Jak se vytvářejí z jednotlivých poznatků znalosti? Je paradoxní, že známe lépe fyziologické funkce jednotlivých součástí lidského organismu než fungování našeho vědomí a zákonitosti procesů, které je tvoří.

Výzkumné aktivity v UI jsou založeny na implicitním předpokladu, že *lidské myšlení je určitá forma výpočtu, který je možné identifikovat, a následně automatizovat.* Zatím se ovšem ukazuje, že taková identifikace nebo alespoň kvantifikace lidské intelektuální schopnosti je velmi obtížná. Inteligence se často opírá o vágní formy znalostí, využití zdravého rozumu, intuice, heuristiky, které nelze plně algoritmicky postihnout. Přitom řešení úloh „hrubou silou“, k němuž

by mohlo svádět spoléhání na výpočetní kapacity současné (či budoucí) techniky, vede velice rychle do slepé uličky, jak dokumentují výsledky v oblasti složitosti uvedené v kap. 9.

Při řešení úloh z reálného světa se běžně pracuje jen s těmi objekty, vztahy a znalostmi, které jsou pro danou úlohu relevantní – prostřednictvím abstrakce se musí vytvořit vhodný model. Při návrhu modelu je možné rozlišit:

- **konceptuální model** – určuje rozsah relevantního „mikrosvěta“, tedy objekty včetně funkčních, popisných a kvantitativních vlastností
- **reprezentační model** – stanoví obecné zásady reprezentace znalostí obsažených v konceptuálním modelu
- **implementační model** – převádí reprezentační model do zpracovatelné podoby (např. ve formě počítačových datových struktur).

Z inženýrského hlediska je podstatné stanovení reprezentačního modelu takovým způsobem, který umožní snadný návrh a zpracování implementačního modelu. Tím se stává klíčovou otázka reprezentace znalostí, pro kterou se rozlišují dvě základní schémata:

- **Procedurální reprezentace** – je zaměřena na otázku „jak“, spočívá v zaznamenání akcí, operací nebo důsledků. Může se opírat např. o aparát formálních gramatik a implementuje se pomocí procedurálních systémů nebo produkčních (na pravidlech založených) systémů.
- **Deklarativní reprezentace** – zaměřuje se na otázku „co“, spočívá v zaznamenání faktů nebo konstatování. Při relačním přístupu může používat např. stromy, grafy nebo sémantické sítě, logický přístup využívá především predikátové logiky.

Při modelování určitého konceptu je samozřejmě možné zvolit různé způsoby reprezentace (např. klausule, formule predikátové logiky, seznamy, matice, atd.). Je-li již určitá reprezentace modelu zvolena, dostávají konkrétní náplň pojmy jako **stav systému**, **stavový prostor**, apod. Řešenou úlohu pak představuje zadaný počáteční stav systému, požadovaný cílový stav a množina akcí, jimiž je možné měnit stav systému. Úkolem je nalézt posloupnost akcí, pomocí nichž lze přejít od počátečního do koncového stavu. Taková posloupnost se nazývá **plánem** nebo také **řešením** dané úlohy.

Stav systému nebo **popis stavu** obsahuje v rámci zvolené reprezentace souhrn všech údajů, které popisují všechny složky modelu. **Stavový prostor** je množina všech stavů systému společně s **operátory** nebo **akcemi**, které způsobují přechody mezi těmito stavy. Stavový prostor je tedy možné chápat jako orientovaný graf, v němž uzly představují stavy systému a hrany vyjadřují přechody. Cílových stavů může být více, kromě explicitního popisu mohou být zadány také podmínkou, kterou musí splňovat. Plánu řešení pak v této interpretaci odpovídá orientovaná cesta z počátečního stavu (uzlu) do nějakého koncového stavu (uzlu).

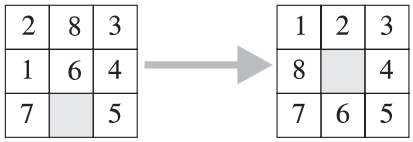
Jelikož problematice hledání cest jsme se dostatečně věnovali již v kapitolách 6 a 7, je nutné objasnit, v čem je specifičnost problematiky hledání řešení ve stavovém prostoru. První rozdíl spočívá v obrovském (někdy i nekonečném) rozsahu stavového prostoru, který u praktických úloh vylučuje explicitní zobrazení odpovídajícího grafu. Prohledávaný graf je pak zadán **implicitně**, zprav. nějakou procedurální reprezentací jeho relace následování. K nalezení cesty je pak třeba použít takových postupů, které při rozumných nárocích na výpočetní prostředky (čas a paměť) projdou a explicitně vyjádří jen velmi omezenou část grafu.

Druhým rozdílem je, že při prohledávání může být k dispozici nějaká doplňující specifická informace, jejímž využitím lze postup směrovat, a vyhnout se tak prohledávání neperspektivních částí grafu. Třetí odlišností je skutečnost, že nalezením cesty (plánu řešení) nemusí úloha skončit, neboť na ně může navazovat jednorázový nebo opakovaný průchod cestou (provedení plánu). S ohledem na povahu této fáze je pak např. možné preferovat rychlé nalezení jakékoliv cesty před zdlouhavým hledáním cesty nejkratší.

To je stručná charakterizace problematiky **řešení úloh (problem solving)**, tedy podoblasti UI, do níž spadají algoritmy, kterými se v této kapitole budeme zabývat.

Příklad 10.1: (Hra „8“) Abychom obecným pojmům dali konkrétnější obsah, uvedeme jednoduchý příklad. Jedná se o známou hru, při které je na desce rozdělené na 3×3 polí umístěno 8 kostek očíslovaných od 1 do 8. Jedno pole zůstává volné a smyslem hry je prostřednictvím posunů kostek docílit toho, aby se od náhodného počátečního uspořádání kostky seřadily do cílového pořadí (viz obr. 10.1).

Každý stav řešení úlohy je zde určen polohou kostek, stavový prostor obsahuje celkem $9! = 362880$ možných stavů úlohy. V každém stavu jsou možné minimálně dva a maximálně čtyři různé posuny, změny stavů jsou reverzibilní. Na obr. 10.2 ukazujeme řešení jednoduchého zadání této hry.



Obrázek 10.1: Hra „8“

Neinformované metody prohledávání

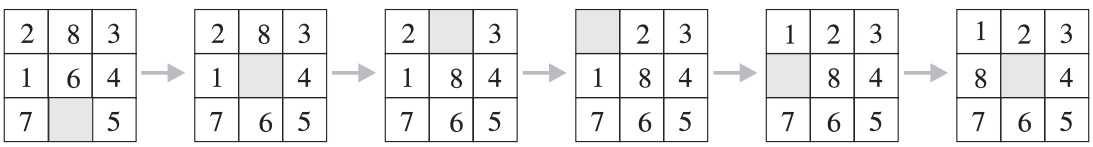
Při hledání libovolné cesty mezi dvěma uzly je možné použít mírně upraveného algoritmu prohledávání grafu do šířky nebo do hloubky (viz algoritmy 4.13 a 4.21). Při této úpravě odpadne úvodní cyklus nastavující všechny uzly grafu na NOVÉ – u implicitně zadaného grafu takové budou všechny dosud nevygenerované uzly a reprezentace grafu se vytváří dynamicky v průběhu hledání. Každý uzel představuje dynamická proměnná vytvořená voláním funkce CREATE_NODE, která vrací ukazatel na záznam uzlu, do něhož uložila (do položky *stav*) hodnotu odpovídajícího stavu. Položka *pred* určuje předchůdce uzlu a položka *open* rozlišuje OTEVŘENÉ a ZAVŘENÉ uzly.

V algoritmu prohledávání do šířky se také upraví podmínka ukončení hlavního cyklu, do které se přidá test nalezení cílového stavu. Další úprava se týká základního kroku obou algoritmů – přechodu k následníkům uzlu. Následníky je třeba získat procedurálně použitím všech operátorů aplikovatelných na daný stav, tento krok se označuje jako **expanze uzlu**. Algoritmus prohledávání do šířky, jehož vstupy jsou relace následování Γ, počáteční stav s a cílový stav t, pak bude mít následující podobu:

Algoritmus 10.2 Prohledávání do šířky

BFS (Γ, s, t)

1	$ps :=$ CREATE_NODE(s)	Vytvoří datovou strukturu
2	$ps\uparrow . open :=$ TRUE; $ps\uparrow . pred :=$ NIL	s uzlem s jako kořenem
3	INIT_QUEUE; ENQUEUE(ps); $Found :=$ FALSE	a uloží jej do fronty.
4	**while** not (EMPTY_QUEUE or $Found$) **do**	Dokud fronta není prázdná,
5	$pu :=$ QUEUE_FIRST	vybere z ní první otevřený uzel.
6	**if** $pu\uparrow . stav = t$ **then** $Found :=$ TRUE	Je-li cílový, končí se.
7	**else for** „každé $v \in \Gamma(u)$“ **do**	Jinak se uzel expanduje
8	**if** „v je nový stav“ **then**	a jeho sousedé,
9	$pv :=$ CREATE_NODE(v)	kteří jsou noví,
10	$pv\uparrow . open :=$ TRUE	se otevřou
11	$pv\uparrow . pred := pu$	s předchůdcem u
12	ENQUEUE (pv)	a uloží do fronty.
13	DEQUEUE; $pu\uparrow . open :=$ FALSE	Odebere a zavře uzel u z fronty
14	**return** $Found$	Vrací příznak úspěchu.



Obrázek 10.2: Řešení konkrétního zadání hry „8“

Časová složitost tohoto algoritmu závisí na složitosti dílčích operací a na velikosti prohledané části stavového prostoru. Pro operaci CREATE_NODE a všechny operace s frontou můžeme předpokládat konstantní čas, stejně jako pro generování jednoho následníka. Složitější operací je zjištění, zda generovaný stav je nový – v závislosti na struktuře reprezentace stavu bychom patrně uplatnili implementaci formou rozptýlené tabulky. Za těchto předpokladů je možné počítat s výslednou asymptotickou složitostí $O(m + n)$, kde m je počet hran a n počet uzlů generované části stavového prostoru.

Lineární složitost prohledávání není možné zlepšit, a tak se varianty prohledávacích algoritmů zaměřují především na dosažení nízkých hodnot m a n. Asymptotická paměťová složitost algoritmu je $O(n)$, skutečné paměťové nároky podstatně závisí na způsobu zakódování stavů. Při implementaci algoritmu je vhodné ukončit hlavní cykl s neúspěchem po vygenerování předem stanoveného maximálního počtu uzlů.

Je-li cílový stav jediný a je navíc explicitně zadán, je možné prohledávání vést směrem od cílového k počátečnímu stavu. Tento postup je ovšem použitelný pouze tehdy, je-li možné určit předchůdce každého stavu (to platí např. v případě reverzibilních operátorů, které má hra „8“), od dopředného hledání se ovšem nijak zásadně neliší. Určité zlepšení lze očekávat při použití **algoritmu obousměrného prohledávání**, při němž se postupuje současně od počátečního i cílového stavu. Při postupu je možné směr pravidelně střídat nebo volit uzel pro expanzi podle nějakého jednoduchého kritéria, které má naději snížit počet generovaných uzlů. Příkladem je expanze uzlu z té fronty otevřených uzlů, která je kratší – na menší šířce se postup směruje více vpřed.

Chceme-li stavový prostor prohledávat do hloubky místo do šířky, stačí ve výše uvedeném algoritmu nahradit frontu otevřených uzlů zásobníkem. Tato varianta by se od předchozí nelišila v tom, že po skončení máme formou p-stromu explicitně uloženou celou vygenerovanou část stavového prostoru. Při prohledávání do hloubky je však možné průběžně „zapomínat“ neúspěšně prohledanou část stavového prostoru a pamatovat si pouze uzly na právě prodlužované cestě. Paměťové nároky se tak dramaticky sníží, pochopitelně za cenu zhoršení časové složitosti. Dalším vhodným doplňkem této varianty prohledávání je omezení hloubky hledání, které zabrání beznadějně se vydat směrem od cíle a skončit vyčerpáním dostupné paměti.

Na rozdíl od prohledávání do šířky nenalezne se při prohledávání do hloubky nejkratší cesta. Tuto nevýhodu odstraňuje tzv. **algoritmus iterativního prohlubování** označovaný **DFID** (depth-first iterative deepenning). Je to vlastně cyklicky opakované hledání s omezenou hloubkou, která se v každém kroku zvyšuje o 1. Prohledávání do šířky a do hloubky (bez omezení hloubky, příp. s iterativním prohlubováním) patří mezi **úplné algoritmy** hledání, neboť zaručují nalezení cesty k cíli, pokud nějaká existuje. Nepříjemnou vlastností všech neinformovaných algoritmů je ovšem skutečnost, že zprav. expandují mnoho uzlů vzdálených od vlastní hledané cesty.

Informované metody prohledávání

Při prohledávání stavového prostoru úlohy, o jejíchž vlastnostech nemáme žádnou pomocnou informaci, nemůžeme expanzi stavů nijak cílevědomě směrovat. Jinak je tomu v případě, že charakter úlohy dovolí definovat nezápornou **hodnotící funkci** f stavů, jejíž nízké hodnoty vyjadřují blízkost k cílovému stavu. Hodnotící funkce tedy dovoluje expandovat jen nejperspektivnější stavy, a tak postupovat žádoucím směrem. Rychlost postupu ovšem podstatně závisí na kvalitě hodnotící funkce – čím kvalitnější heuristiky pro řešení úlohy se v ní uplatní, tím efektivněji dokáže vést hledání k cíli.

Nejjednodušší formu využití hodnotící funkce představuje **gradientní algoritmus (hill-climbing)**, který vychází z prohledávání do hloubky. Při expanzi uzlu se následníci seřadí podle hodnotící funkce a do zásobníku otevřených uzlů se uloží tak, že na vrcholu bude nejperspektivnější z nich. Díky tomu postupuje hledání směrem k extremní hodnotě – může se ovšem jednat jen o extrém lokální, který je stále od cíle vzdálen.

Jiný způsob využití hodnotící funkce představuje **algoritmus paprskového prohledávání (beam-search)**, který je variantou prohledávání do šířky. V každé hladině uzlů stejně vzdálených od počátku se však do fronty otevřených uzlů zařadí nejvýše k nejlepších kandidátů expanze. Tento postup dovoluje udržet počet generovaných uzlů v rozumných mezích, zároveň však může selhat, pokud je možné dospět k cíli pouze přes uzel vynechaný v některé hladině.

Systematičtějším způsobem využívá hodnotící funkce **algoritmus uspořádaného prohledávání**. Pracuje podobně jako gradientní algoritmus, ale otevřené uzly ukládá do prioritní fronty podle jejich ocenění hodnotící funkcí. K expanzi se pak vybírá otevřený uzel s nejmenší hodnotou f – ale to je vlastně nám dobře známý Dijkstrův algoritmus, v němž se namísto (odhadované) vzdálenosti od počátku používá právě hodnotící funkce f! Je-li výpočet hodnoty $f(u)$ nějak závislý na hodnotě f pro předchůdce uzlu, je nutné využít i náležitě upravenou operaci relaxace hrany a upravovat pro dříve generované uzly jejich ohodnocení a zařazovat je znovu do fronty otevřených uzlů.

Až dosud jsme předpokládali, že hodnotící funkce je nějakou ad hoc stanovenou mírou přiblížení k cílovému stavu. Tato funkce ale není zcela spolehlivá a řídit prohledávání pouze podle ní může způsobit, že se expandují uzly velmi vzdálené jak od počátku, tak i od cíle. Výhodné je tedy uvažovat hodnotící funkci ve tvaru

$$f(u) = g(u) + h(u), \tag{10.1}$$

kde $g(u)$ představuje vzdálenost uzlu u od počátku s a $h(u)$ vzdálenost od u k cíli t. Hodnotící funkce pak vyjadřuje délku cesty řešení procházející uzlem u. Vzdálenost zde (podobně jako u Dijkstrova algoritmu) chápeme v obecném smyslu, ne nutně jako počet hran nejkratší cesty. Operátory úlohy mohou mít totiž přiřazeny různé ceny, a tak bude odpovídající stavový prostor vyjádřen orientovaným grafem s ohodnocenými hranami. Algoritmus uspořádaného prohledávání s takto koncipovanou hodnotící funkcí se nazývá **algoritmus A**.

Cvičení

10.1-1. Při návrhu algoritmu obousměrného prohledávání jsme v obou směrech předpokládali uplatnit prohledávání do šířky. Zvažte, zda by bylo účelné v jednom nebo obou směrech postupovat do hloubky.

10.1-2. Předpokládejme následující zjednodušenou variantu gradientního algoritmu: po každé expanzi se vybere z následníků uzel s nejmenší hodnotou hodnotící funkce, všechny ostatní uzly včetně právě expandovaného se zapomenou. Srovnejte výhody a nevýhody této varianty oproti původní verzi gradientního algoritmu.

10.2 Heuristické hledání

Algoritmus A tedy prodlouží takovou cestu, která se v daném okamžiku zdá být součástí nejkratší cesty řešení. Je ovšem třeba vyřešit podstatný detail – jak se mají počítat hodnoty $g(u)$ a $h(u)$ použité ve vztahu (10.1) definujícím hodnotící fukci. V praktických úlohách neznáme pravidla přesného výpočtu těchto hodnot, neboť jinak bychom patrně byli schopni úlohu řešit přímo a nikoliv pomocí hledání.

Namísto přesných hodnot $g(u)$ a $h(u)$ se tedy musíme spokojit s aproximacemi: $\hat{g}(u)$ je odhad vzdálenosti (ceny přechodu) od počátku do uzlu u a $\hat{h}(u)$ je odhad vzdálenosti od u k cíli t. Zatímco za hodnoty $\hat{g}(u)$ je možné použít průběžně zpřesňovanou hodnotu vzdálenosti počítanou jako v Dijkstrově algoritmu, výpočet hodnot $\hat{h}(u)$ je zcela závislý na existenci nějakých kvantifikovatelných heuristických pravidel, která by dokázala odhadovat vzdálenosti v dosud nevygenerované části stavového prostoru. Funkce $\hat{h}(u)$ je tedy numerickým vyjádřením naší heuristické informace, a tak se označuje jako **heuristická funkce**. Algoritmus A, který se

opírá o aproximovanou hodnotící funkci

$$\hat{f}(u) = \hat{g}(u) + \hat{h}(u),$$

budeme nazývat **algoritmem A***.

Až dosud jsme při hledání cesty řešení nebrali ohled na výslednou délku nalezené cesty. Pokud je součástí zadání požadavek nalézt nejkratší cestu, musíme zjistit, zda a za jakých předpokladů zajistí použití určitého algoritmu splnění této podmínky. Je např. zřejmé, že prohledáváním do šířky (popř. obdobou Dijkstrova algoritmu pro ohodnocené operátory) se nalezne nejkratší cesta, při prohledávání do hloubky to není zaručeno. Algoritmus, který zaručuje nalezení nejkratší cesty řešení (pokud vůbec nějaká cesta existuje), se nazývá **přípustný algoritmus prohledávání**. Všimneme si nyní podmínek přípustnosti algoritmu A*, který pro úplnost vyjádříme ve zjednodušené podobě.

Algoritmus 10.3 Algoritmus uspořádaného hledání

ALG_A*(Γ, s, t)

1	$S := \emptyset$	Zatím žádný zavřený uzel.
2	INIT-QUEUE(Q)	Do fronty zařaď jen
3	ENQUEUE(Q, s); $Found :=$ FALSE	počáteční uzel s.
4	**while** not (EMPTY(Q) or $Found$) **do**	Dokud zbývá otevřený uzel,
5	$u :=$EXTRACT-MIN(Q)	vyber uzel s nejmenším $f(u)$
6	$S := S \cup \{u\}$	a přeřaď jej mezi uzavřené.
7	**if** $u = t$ **then** $Found :=$ TRUE	Je-li cílový, konec výpočtu.
8	**else for** každý uzel $v \in \Gamma(u)$ **do**	Jinak pro všechny následníky,
9	**if** „v je nový uzel“	kteří jsou noví,
10	**then** „zařaď v do Q s vypočtenou hodnotou $f(v)$“	
11	**else** „přepočti $g(v)$ a je-li nová hodnota $f(v)$ menší,	
12	zařaď v do Q s novou hodnotou $f(v)$”	
13	DEQUEUE	Odeber uzel u z fronty.
14	**return** $Found$	Vrať příznak úspěchu.

Věta 10.4: Nechť platí $\hat{h}(u) \leq h(u)$ pro každý uzel u stavového prostoru. Potom pro libovolnou nejkratší cestu řešení P existuje kdykoliv před ukončením činnosti algoritmu A* otevřený uzel u' cesty P takový, že

$$\hat{f}(u') \leq f(s).$$

Důkaz: Zvolme jako u' první otevřený uzel cesty P ve směru od s. Cesta P je nejkratší, takže pro libovolný její uzel x platí $f(x) = g(x) + h(x) = d(s,x) + d(x,t) = d(s,t)$. Je tedy

$$f(s) = f(u') = g(u') + h(u') \geq \hat{g}(u') + \hat{h}(u') = \hat{f}(u').$$

Při přechodu k aproximovaným hodnotám jsme pro $\hat{h}(u')$ použili předpokladu věty a rovnosti $\hat{g}(u') = g(u')$ plynoucí z toho, že všechny předchozí uzly na optimální cestě P byly už uzavřeny, takže je určena definitivní hodnota vzdálenosti $d(s,u')$. △

Důsledek : Platí-li předpoklady předchozí věty, pak je pro každý uzel u uzavřený algoritmem A splněna nerovnost $\hat{f}(u) \leq f(s)$.

Věta 10.5: Nechť platí $\hat{h}(u) \leq h(u)$ pro každý uzel u stavového prostoru a nechť je ohodnocení každého operátoru nejméně rovno jistému reálnému číslu $\delta > 0$. Potom je algoritmus A* přípustný.

Důkaz: Postupujeme sporem, přitom předpokládáme existenci cesty řešení. Pokud A* neskončí nalezením nejkratší cesty, můžeme rozlišit následující tři případy:

1. A* skončí hledání na jiném než cílovém uzlu
2. A* nikdy neskončí svoji činnost (zacyklí se)
3. A* skončí hledání na cílovém uzlu, ale nenalezne nejkratší cestu.

Ad 1. Hlavní cyklus algoritmu skončí pouze buď vyčerpáním fronty otevřených uzlů nebo nalezením cíle. Fronta ale nemůže být prázdná, neboť by to bylo ve sporu s větou 10.4.
Ad 2. Jelikož δ je dolní mez ohodnocení hran grafu a vzdálenost k cíli je rovna $f(s)$, může nejkratší cesta řešení obsahovat nejvýše $k = f(s)/\delta$ hran. Současně platí, že pro každý uzel u, k němuž vede z s nejkratší cesta tvořená alespoň $(k+1)$ hranami, bude

$$\hat{f}(u) = \hat{g}(u) + \hat{h}(u) \geq (k+1)\delta = k\delta + \delta > f(s).$$

Algoritmus A* tedy díky větě 10.4 expanduje nejvýše uzly nacházející se k hran od uzlu s – takových je ale konečně mnoho a konečný je i počet různých cest, které k nim z s vedou a mohou způsobit opakované otevření uzlu s novou hodnotou $\hat{f}(u)$. Algoritmus A* se tedy nemůže zacyklit.
Ad 3. Pokud algoritmus A* skončí nalezením cíle t s hodnotou $\hat{f}(t) = \hat{g}(t) > f(s)$, je to ve sporu s větou 10.4, neboť v okamžiku výběru uzlu t byl k dispozici jiný uzel u' s hodnotou $\hat{f}(u') \leq f(s)$. △

Algoritmus A* je tedy přípustný za poměrně slabých předpokladů o použité heuristické funkci $\hat{h}(u)$. Dlužno však podotknout, že v praktických úlohách nelze často ověřit ani takto jednoduchou podmínku. Triviální volba $\hat{h}(u) = 0$ zaručuje přípustnost, ale v tomto případě dostáváme neinformovaný Dijkstrův algoritmus. Intuitivně je jasné, že větší efekt pro hledání má heuristická funkce, která se co nejvíce zdola přibližuje k ideální funkci $h(u)$.

Definice 10.6: Algoritmus A_1^* používající heuristickou funkci $\hat{h}_1$ nazýváme **lépe informovaným** než algoritmus A_2^* používající heuristickou funkci $\hat{h}_2$, pokud platí $\hat{h}_1(u) \geq \hat{h}_2(u)$ pro každý stav u stavového prostoru.

Pokud jsou obě heuristické funkce přípustné, oba algoritmy naleznou nejkratší cestu řešení. Očekávali bychom však, že lépe informovaný algoritmus k tomu bude muset expandovat méně (nebo nejvýše stejně tolik) stavů, než algoritmus informovaný hůře. Pro takové očekávání je však třeba požadavky na heuristické funkce dále zesílit.

Definice 10.7: Heuristickou funkci $\hat{h}(u)$ nazýváme **konzistentní**, pokud pro všechny stavy u, v platí

$$\hat{h}(u) - \hat{h}(v) \leq d(u, v), \tag{10.2}$$

kde $d(u, v)$ je přesná vzdálenost stavů u, v ve stavovém prostoru.

Konzistence je opravdu silnějším požadavkem – předpokládáme-li nulovou hodnotu heuristické funkce v cílovém stavu, pak pro konzistentní heuristickou funkci platí

$$\hat{h}(u) - \hat{h}(t) \leq d(u, t) \quad \Rightarrow \quad \hat{h}(u) - 0 = \hat{h}(u) \leq h(u).$$

Věta 10.8: Nechť algoritmus A* používá konzistentní heuristickou funkci h. Potom pro každý uzel v uzavřený algoritmem A* platí v okamžiku jeho uzavření $\hat{g}(v) = g(v)$.

Důkaz: Postupujeme sporem – nechť je $\hat{g}(v) > g(v)$. Nechť P je nějaká nejkratší cestu z s do v, označme u první její otevřený uzel ve směru od s. Všichni předchůdci uzlu u na cestě P jsou již uzavřeni, takže platí $\hat{g}(u) = g(u)$. Podle lemmatu 6.2 pro cestu P bude

$$g(v) = d(s, v) = d(s, u) + d(u, v) = g(u) + d(u, v) = \hat{g}(u) + d(u, v),$$

takže z předpokladu $\hat{g}(v) > g(v)$ a konzistence funkce $\hat{h}$ dostáváme

$$\hat{g}(v) > \hat{g}(u) + d(u, v) \quad \Rightarrow \quad \hat{g}(v) + \hat{h}(v) > \hat{g}(u) + d(u, v) + \hat{h}(v) \geq \hat{g}(u) + \hat{h}(u).$$

To ale znamená, že $\hat{f}(v) > \hat{f}(u)$, což je ve sporu s tím, že pro uzavření byl vybrán uzel v a ne uzel u. △

Věta 10.9: Nechť jsou A_1^*, A_2^* přípustné algoritmy takové, že A_1^* je lépe informovaný než A_2^* a používá konzistentní heuristickou funkci $\hat{h}_1$. Potom platí, že jestliže algoritmus A_1^* expanduje nějaký uzel u, potom jej také expanduje algoritmus A_2^*.

Důkaz: – viz [36]. △

Jestliže ověření přípustnosti heuristické funkce bylo obtížné, pak je potvrzení konzistence prakticky vyloučeno. Uvedené teoretické výsledky mají tedy jen velmi omezený praktický význam. Konkrétnější odhady počtu uzlů expandovaných při použití algoritmu A^* je možno získat za velmi zjednodušujících předpokladů o struktuře stavového prostoru (viz např. [33]).

Popsanými algoritmy se zdaleka nevyčerpávají všechny vyvinuté varianty heuristického prohledávání. Existuje např. obusměrná verze algoritmu A^* nazvaná **SOH (symetrické obousměrné hledání)** (viz [20]), která je přípustná a kterou lze docílit nižšího počtu expandovaných uzlů než při prohledávání jednostranném. Zajímavá je rovněž varianta iterativního prohlubování **IDA* (iterative deepening A*)**, při níž se na počátku nastaví prahová hodnota hodnotící funkce a uzly s vyšší hodnotou se neuchovávají. Pokud hledání neuspěje, nový práh se nastaví na nejblíže vyšší hodnotu dosaženou v minulém průchodu. Podrobnější přehled základních metod hledání lze nalézt v [25], popis zahrnující i víceprocesorové metody je k dispozici např. v [28].