Traveling Umpire Problem

Adam Matuš (xmatus31@stud.fit.vutbr.cz)

Duben 2021

1 Úvod a definice problému

Traveling Umpire Problem (TUP) je problém z oblasti plánování rozvrhů sportovních událostí, jako jsou například turnaje Major League Baseball a podobné. Této problematice se věnuje mnoho prací z operačního výzkumu. V tomto projektu je TUP řešen pomocí genetického algoritmu, který byl publikován v článku od autorů Michael A. Trick a Hakan Yildiz [1].

Instance problému o velikosti n obsahuje n rozhodčích, $2n$ týmů a $4n - 2$ časových slotů. Každý časový slot obsahuje n her a každá hra je reprezentovaná dvojicí (i, j), kde i je domácí tým a j jsou hosté. Z těchto faktů lze odvodit, že v každém časovém slotu hrajou všechny týmy a jsou zaměstnaní všichni rozhodčí. Primárním cílem TUP je najít takový plán, kde celková uražená vzdálenost rozhodčí mezi místy konání her je co nejmenší, přičemž musí být splněna určitá omezení:

(1) Každá hra musí mít přiřazeného rozhodčího.

(2) Každý rozhodčí je přiřazen právě jedné hře v daném časovém slotu.

(3) Každý rozhodčí navštíví každý domácí tým alespoň jednou.

(4) Žádný rozhodčí nenavštíví domácí tým vícekrát než $n - d_1$ slotů po sobě.

(5) Žádný rozhodčí nesmí vidět tým vícekrát než $\lfloor \frac{n}{2} \rfloor - d_2$ slotů po sobě.

Parametry d_1 a d_2 určují striktnost omezení (4) a (5). Nastavení parametrů na $d_1 = 0$, $d_2 = 0$ je nejstriktější možné, a naopak $d_1 = n - 1$, $d_2 = \lfloor \frac{n}{2} \rfloor - 1$ úplně odstraní jejich efekt.

V následující kapitole 2 jsou představeny metody publikované v článku. V kapitole 3 bude implementace otestovaná a srovnaná na různých instancích problému. Experimenty se potom zaměřují na různé drobné úpravy algoritmu a parametrů genetického algoritmu. Kapitola 4 shrnuje výsledky projektu.

2 Řešení a implementace

V tomto projektu je implementovaný genetický algoritmus (GA) převzatý z publikace [1]. GA je založený na iterativním vytváření populace řešení, kde cílem je zlepšovat průměrnou kvalitu řešení v každé další generaci. Počáteční populace není generovaná zcela náhodně, ale pomocí heuristiky Greedy Matching. Další generace jsou potom vytvářeny aplikacemi operátorů křížení a mutace na náhodně zvolené jedince v populaci. Kvalita řešení je hodnocena funkcí *fitness*, která počítá celkovou vzdálenost řešení.

Implementační jazyk je Python 3.8. Projekt využívá knihovnu `networkx` pro řešení grafových problémů.

2.1 Reprezentace jedince

Jedno řešení je reprezentované jako kompletní plán pro všechny rozhodčí. Pro instanci o velikosti n je potom plán posloupnost $4n - 1$ slotů, kde každý slot je permutace n her přiřazených v kontrétním časovém slotu. Příklad jednoho řešení pro $n = 3$ je znázorněn v tabulce 1.

Sloty	1	2	3	4	5	6	7	8	9	10
Rozhodčí 1	(1, 5)	(6, 4)	(3, 4)	(2, 4)	(4, 1)	(6, 1)	(1, 3)	(5, 3)	(2, 1)	(4, 6)
Rozhodčí 2	(4, 3)	(3, 5)	(1, 6)	(5, 6)	(6, 3)	(3, 2)	(4, 2)	(2, 6)	(3, 6)	(5, 1)
Rozhodčí 3	(6, 2)	(1, 2)	(5, 2)	(3, 1)	(2, 5)	(4, 5)	(6, 5)	(1, 4)	(5, 4)	(2, 3)

Tabulka 1: Příklad jednoho optimálního řešení pro instanci s 3 rozhodčími a 6 týmy

2.2 Výchozí populace

Před samotným startem genetického algoritmu je potřeba vygenerovat vhodnou populaci, například pomocí heuristiky Probabilistic Greedy Matching (PGM). Základním principem algoritmu je začít prvním slotem plánu a přiřadit následující slot s takovou permutací, aby nebyla porušena žádná omezení kromě (3) a zároveň byla vzdálenost co nejkratší. Aby vznikala různorodá populace, jsou náhodně občas vybraná i neoptimální přiřazení. Hledání přiřazení je řešeno pomocí bipartitního grafového algoritmu *minimum weight full matching*, kde již zvolený částečný plán figuruje jako levé uzly grafu a následující hry v slotu jsou pravé uzly.

Ne vždy je možné tímto způsobem snadno nalézat řešení, které neporušují omezení. Tato situace není v původním článku přesně popsaná, takže je v tomto projektu zvoleno náhodné váhování nevhodných hran. Váhy jsou zvoleny dostatečně velké, aby penalizovaly porušená omezení. Při testování algoritmu se ukázalo, že i tento způsob generuje kvalitní a různorodou výchozí populaci.

2.3 Operátor křížení

Běžně se v genetických algoritmech využívá například jednoduché jednobodové křížení, kdy je náhodně zvoleno místo křížení a části na obou stranách jsou mezi rodiči vyměněné,

čímž vznikne nový jedinec. V tomto algoritmu je použitý vylepšený operátor s lokální optimalizací v místě křížení, aby nový jedinec měl nejlepší možnou fitness.

Principem je vzít levou část plánu prvního rodiče a nalézt nejlepší možnou permutaci pro pravou část plánu druhého rodiče. K této optimalizaci se opět používá grafový algoritmus *minimum weight full matching* s tím rozdílem, že pravé uzly nejsou jen hry z následujícího slotu, ale celá pravá část plánu. Tato optimalizace také již bere na vědomí všechny omezení. Porušení je tvrdě penalizované vysokou hodnotou hrany v grafu.

2.4 Operátor mutace

Z populace jsou náhodně vybráni jedinci, na které se aplikuje mutace. Mutace jedince náhodně vybere jeden časový slot a prohodí 2 náhodně vybrané hry. Mutace pomáhá genetickému algoritmu udržovat různorodost populace a případně objevovat řešení, na které křížení nestačí.

2.5 Evaluace řešení

Každé řešení v populaci je možné ohodnotit podle uražené vzdálenosti všech rozhodčí. Skóre fitness řešení je součet vzdáleností z místa konání hry prvního slotu do místa v posledním slotu, a to pro všechny rozhodčí. Za každé porušení podmínek je k fitness přičtena vysoká penalizace, která násobně převyšuje fitness hodnotu počítánou z uražené vzdálenosti. Tímto je vystaven tlak na ty jedince populace, kteří mají sice nízkou vzdálenost, ale porušují některá omezení.

2.6 Genetický algoritmus

Velikost populace je stanovena na 500 jedinců. Na začátku každé generace je pomocí rovnoměrně náhodného křížení vyrobeno 250 nových řešení, přičemž nová řešení která již existují v populaci jsou zahozena. Do další generace se dostane 500 řešení s nejlepší hodnotou fitness. Na každého jedince v populaci je v každé generaci aplikovaný operátor mutace s pravděpodobností 0.05.

3 Testování a experimenty

V této sekci bude implementace porovnaná s výsledky autorů původního algoritmu. Použitá datová sada s různě složitými instancemi problému je volně dostupná na internetu[1]. Testování proběhlo na stroji s operačním systémem Windows 10 a procesorem Intel Core i5-6600K 3.90 GHz. Algoritmus byl u každého běhu ukončen po 30 minutách nebo pokud nalezl známé optimální řešení dříve.

Tabulka 2 srovnává rychlost algoritmu na relativně jednoduchých instancích TUP, u kterých je známé optimální řešení. Čísla v závorkách značí, že nejlepší nalezená vzdálenost je odlišná od původních autorů. U instancích do velikosti 8 jsou výsledky velmi podobné. Instance s 10 týmy jsou viditelně náročnější než předchozí instance,

[1] `http://benchmark.gent.cs.kuleuven.be/tup/en/`

například u instancí 10 a 10C nebylo možné najít optimální řešení. Instanci 10B se naopak daří řešit mnohem rychleji, než autorům původní implementace.

Instance	Opt. vzdálenost	Nejlepší vzdálenost		Čas nalezení [s]	
		Původní	xmatus31	Původní	xmatus31
4	5 176	5 176	5 176	0	1
6	14 077	14 077	14 077	1	1
6A	15 457	15 457	15 457	1	1
6B	16 716	16 716	16 716	1	1
6C	14 396	14 396	14 396	1	1
8	34 311	34 311	34 311	5	4
8A	31 490	31 490	31 490	12	10
8B	32 731	32 731	32 731	5	7
8C	29 879	29 879	29 879	4	4
10	48 942	48 942	(49 132)	180	40
10A	46 551	46 632	46 632	120	110
10B	45 609	45 609	45 609	120	8
10C	43 149	43 149	(43 193)	1200	60

Tabulka 2: Srovnání výsledků s původní publikací, parametry $d_1 = d_2 = 0$

Prvním vlastním experimentem bylo vyzkoušet větší velikosti populace než 500 na složitějších instancích. Tabulka 3 shrnuje výsledky na instancích s 10 týmy. Zvýšení populace pomohlo najít optimální řešení u všech instancí kromě 10A. Za pozornost stojí instance 10C, u které se čas podstatně zlepšil v porovnání s referenčním výsledkem.

V dalších experimentech byly vyzkoušeny různé pravděpodobnosti mutace. Zvyšování pravděpodobnosti většinou vedlo ke zhoršování rychlosti konvergence a k více chaotickému průběhu průměrné fitness populace. Snižování pravděpodobnosti nevedlo k zásadním změnám v chování algoritmu. Při úplném vynechání mutace (pravděpodobnost 0) potom GA nalezl některá řešení o něco rychleji, ale často také uvázl s velmi podobnou populací, kde samotné křížení negenerovalo nová řešení.

Instance	Opt. vzdálenost	Pop. = 1000		Pop. = 2000	
		Vzdálenost	Čas [s]	Vzdálenost	Čas [s]
10	48 942	48 942	914	48 942	154
10A	46 551	46 632	60	46 632	90
10B	45 609	45 609	15	45 609	22
10C	43 149	43 149	75	43 149	100

Tabulka 3: Výsledky pro různé velikosti populace

4 Závěr

V tomto projektu byl úspěšně implementován genetický algoritmus pro řešení Traveling Umpire Problem z referenčního článku [1]. GA dokáže najít kvalitní a pro jednodušší instance i optimální řešení velmi rychle. Implementace dosahuje podobných výsledků jako referenční.

Součástí projektu je implementace jako program v jazyce Python. Manuál ke spouštění je v souboru README.

Reference

[1] Michael A. Trick and Hakan Yildiz. Locally optimized crossover for the traveling umpire problem. *European Journal of Operational Research*, 216(2):286–292, 2012.