패치 우선순위화를 위한 소스 코드 표현방식에 따른 유사도 계산 기법 별 영향도 분석

허진석[O1], 정호현[1], 이은석[2]
성균관대학교 전자전기컴퓨터공학과[1], 성균관대학교 소프트웨어대학[2]
{mrhjs225, jeonghh89, leees}@skku.edu

An Analysis of Similarity Techniques for Patch Prioritization : The Aspect of Source Code Expression Methodologies

Jinseok Heo[O1], Hohyeon Jeong[1], Eunseok Lee[2]
Department of Electrical and Computer Engineering, Sungkyunkwan University[1]
College of Software, Sungkyunkwan University[2]

요 약

자동프로그램수정(Automated Program Repair) 분야에서는 더욱 빠르게 옳은 패치를 선별하기 위해 패치 우선순위화를 활용한다. 이를 위해 기존 기법들은 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트 간의 유사도 점수를 구하여 우선순위화를 한다. 하지만 유사도 계산에 사용되는 소스 코드들의 표현 방식에 따른 우선순위화 성능에 대한 분석 없이 사용되고 있다. 따라서 본 논문에서는 소스 코드 표현 방식에 따라 유사도 기법들이 패치 우선순위화에 얼마나 영향을 미치는지에 대한 분석을 진행한다. 분석 결과 소스 코드의 표현 방식을 추상 구문 트리(Abstract Syntax Tree), 문자열, 벡터로 나누었을 때 벡터로 표현하여 계산했을 때 유사도 기법의 패치 우선순위화 성능이 제일 높았다. 또한 옳은 패치가 새로운 코드를 삽입하는 형태의 패치일 경우 분석 대상의 모든 유사도 기법들의 우선순위화 성능이 낮았다.

1. 서 론

자동프로그램수정(APR, Automated Program Repair) 분야에서는 더욱 짧은 시간에 옳은 패치를 선별하기 위해 패치 우선순위화를 활용한다 [1]. APR 과정은 크게 버그 추적, 패치 생성, 패치 검증 단계로 나뉘게 된다. 패치 우선순위화는 이 중 패치 검증 단계에 적용되는 기법으로 후보 패치 중 사람에 의해 통과될 수 있는 패치인 옳은 패치를 더욱 높은 순위의 검증 대상으로 높이는 것에 목표가 있다.

기존의 패치 우선순위화 기법들은 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트의 유사도를 계산하여 우선순위화를 진행한다 [2, 3, 4, 5]. Ripon K. Saha et al. [2]은 버기 코드가 가진 문맥을 반영하기 위해 의심스러운 스테이트먼트가 속한 라인과 후보 패치가 적용된 스테이트먼트의 라인 전, 후의 3줄을 각각 하나의 컨텍스트로 취급한다. 컨텍스트별로 소스 코드를 토큰화한 후 식별자(identifier) 만을 필터링하여 식별자 토큰 집합들을 구성한다. 그리고 식별자 토큰 집합 간의 Jaccard 계수를 구함으로써 유사도를 계산한다. Xuan-Bach D. Le et al. [3]은 소스 코드를 벡터화하여 벡터 간 코사인 유사도를 구한다. 벡터화에는 추상 구문 트리(AST, Abstract Syntax Tree)의 노드 유형을 기준으로 소스 코드를 벡터화 하였다.

소스 코드 간 유사도를 구할 때 사용되는 소스 코드의 표현 방식에는 크게 세 가지가 존재한다.

■ AST

AST는 소스 코드에서 발생하는 구조를 트리로 나타낸 것으로 AST의 특성인 노드의 개수, 노드 중복 여부, 노드의 유형 등을 통해 유사도를 구한다. Ming Wen et al. [4]은 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트 각각이 속한 메소드들을 AST로 변환하여 노드 유형별 겹치는 수를 계산하여 유사도를 도출한다.

■ 문자열

소스 코드를 단순 문자열로 취급하게 되면 해당 문자열 간 유사도를 계산하여 이를 소스 코드 간 유사도로 활용한다. Zimin Chen and Martin Monperrus [5]는 유사한 소스 코드를 찾기 위해 소스 코드를 문자열 취급하여 공통된 가장 긴 subsequence인 LCS(Longest Common Subsequence)를 구하였다. 그리고 이 길이를 유사도 계산에 사용하였다.

■ 벡터

소스 코드를 벡터화하여 벡터 간의 유사도를 구하게 된다. 벡터 기반 유사도 계산 방식 또한 기존 연구들 [3, 5]에서도 사용되고 있다.

하지만 많은 패치 우선순위화 기법들이 유사도 계산에 사용되는 소스 코드의 표현 방식에 따른 패치 우선순위화 성능의 분석 없이 유사도를 측정하고 있다. 따라서 본 논문에서는 소스 코드의 표현 방식에 따른 유사도 계산 방식들이 패치 우선순위화에 어떠한 영향을 미치는지에 대해 분석한다. 분석을 위해 버그 추적 기법으로는 Ochiai, 패치 생성 및 검증에는

ConFix [6]를 사용하였다. 분석에 사용된 벤치마크는 Defects4j로 이 중 기법 ConFix가 옳은 패치를 생성한 버그들을 대상으로 분석을 진행했다.

본 논문의 기여점으로는

- ✓ 소스 코드의 표현 방식에 따른 유사도 기법 영향력 분석을 하였다.
- ✓ 사람이 작성한 패치 형태에 따른 유사도 기법들의 효과를 확인하였다.

이어지는 2장에서는 분석에 사용되는 유사도 기법에 관해 설명한다. 3장에서는 분석을 위해 설정된 실험 환경과 패치 우선순위화 과정에 대해 자세히 설명한다. 4장에서는 분석 결과에 대해 논의하고 5장에서는 향후 연구와 함께 본 논문을 마무리 짓는다.

2. 유사도 기법

본 장에서는 분석에 사용되는 소스 코드 유사도 계산 기법 6개를 계산의 대상이 되는 소스 코드의 표현 방식에 따라 분류하여 소개한다.

2.1 AST

2.1.1 중복 노드 개수

중복 노드 개수는 Tao Ji et al. [7]에서 사용된 기법으로 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트를 AST로 나타냈을 때 겹치는 노드의 개수를 측정하여 유사도를 계산한다.

$$Sim_{overlap} = \frac{2 * O}{2 * O + S + C} \quad (1)$$

중복 노드 개수에 따른 유사도 계산은 위의 식과 같다. O는 겹치는 노드의 개수 S, C는 각각 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트의 AST 노드들 중 겹치지 않은 노드의 개수를 의미한다.

2.1.2 Genealogy Context

Genealogy context는 Ming Wen et al. [4]에서 제안된 유사도 기법으로 후보 패치가 특정 유형의 노드와 같이 사용되는 특징을 고려하기 위해 제안되었다. AST 노드별 유형을 구분하여 유사도를 계산하며, 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트 각각이 속한 컨텍스트를 대상으로 AST를 만들어 유사도를 계산하며 식은 아래와 같다.

$$Sim_{gen} = \frac{\sum_{t \in T} \min\left(AST_S(t), AST_C(t)\right)}{\sum_{t \in T} AST_C(t)} \quad (2)$$

T는 후보 패치가 적용된 스테이트먼트의 컨텍스트에 등장하는 모든 AST 노드의 유형을 뜻하며, $AST_S(t)$, $AST_C(t)$ 각각 의심스러운 스테이트먼트 와 후보 패치가 적용된 스테이트먼트의 컨텍스트 AST에 등장하는 특정 유형 t에 해당하는 노드의 개수이다.

2.2 문자열

2.2.1 LCS

LCS는 두 문자열 간 공통된 가장 긴 subsequence를 의미하며, 문자를 기반으로 유사도를 측정하기 때문에 가장 기본적인 syntactic 유사도 기법이다 [5]. 이를 기반으로 Zimin Chen and Martin Monperrus [5]은 소스 코드 간 유사도를 측정하였다.

$$Sim_{lcs} = \frac{len\left(LCS(string_S, string_C)\right)}{\max\left(len(string_S), len(string_C)\right)} \quad (3)$$

$string_S$와 $string_C$는 각각 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트를 문자열로 변환한 것이다. len 은 해당 문자열의 길이를 의미한다.

2.2.2 Contextual Similarity

Contextual similarity는 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트 간 유사도를 구할 때 근처 코드를 고려하여 유사도를 계산하기 위해 제안된 방식이며 Ripon K. Saha et al. [2] 에서 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트간 유사도를 구하기 위해 사용되었다.

근처 코드를 고려하기 위해 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트의 전, 후 3줄을 각각 하나의 컨텍스트로 취급하였고 해당 컨텍스트를 단어 단위로 토큰화하였다. 이후 토큰들 중 식별자만을 선별하여 토큰 집합인 S_S, S_C를 생성한 후 Jaccard 계수를 통해 유사도를 구한다.

$$Sim_{context} = \frac{|S_S \cap S_C|}{S_S \cup S_C} \quad (4)$$

2.3 벡터

2.3.1 코사인 유사도

코사인 유사도는 두 벡터가 가르키는 방향이 얼마나 유사한지를 의미한다. 코사인 유사도 계산 식은 아래와 같다.

$$Sim_{cos} = \frac{\vec{S} \cdot \vec{C}}{|\vec{S}| * |\vec{C}|} \quad (5)$$

S와 C는 각각 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트의 벡터 형태를 의미한다.

2.3.2 유클리디안 유사도

유클리디안 거리는 n 차원의 공간에서 두 점 간의 거리를 계산하는 방식이다. 본 논문의 분석에서는 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트가 n 차원 공간에서 얼마나 가까운지를 측정하는 방식으로 취급하였다. 의심스러운 스테이트먼트와 후보 패치가 적용된 스테이트먼트에 해당하는 벡터를 $S = (S_1, S_2, S_3 \dots\dots, S_n)$ 와 $C = (C_1, C_2, C_3 \dots\dots, C_n)$로 취급할 때 계산 식은 아래와 같다.

$$Ed = \sqrt{\sum_{i=1}^{n} (S_i - C_i)^2} \quad (6)$$

$$Sim_{ed} = \frac{1}{1 + Ed} \quad (7)$$

Ed는 유클리디안 거리를 뜻하며 0과 1 사이의 값으로 정규화하기 위해 두 번째 식을 통해 최종 유사도 점수를 계산한다.

표 1 데이터 셋

프로젝트	버그 ID	버그 수
Chart	1, 10, 11, 24	4
Closure	14, 38, 73, 92, 109	5
Lang	6, 24, 26, 51, 57	5
Math	5, 30, 33, 34, 70, 75	6
Time	19	1
합계	-	21

3. 분석 방법

3장에서는 분석에 사용된 도구들과 패치 우선순위화 과정, 분석에 사용된 데이터 셋과 실험 세팅에 대해 자세히 설명한다.

3.1 구현

APR 과정 구현을 위해 버그 추적 과정은 Ochiai 기법을 사용하였고 패치 생성 및 검증 과정은 도구 ConFix를 사용하여 구현하였다. AST 탐사 및 JDT에 내장되어있는 파서(parser)를 사용하였고 소스 코드 벡터화를 위해 도구 Deckard [8]의 알고리즘을 구현하여 벡터화를 진행하였다.

3.2 패치 우선순위화 과정

패치 우선순위화는 패치 생성 과정과 패치 검증 과정 사이에 구현하였다. 패치 생성을 통해 생성된 후보 패치를 적용한 스테이트먼트들을 대상으로 의심스러운 스테이트먼트와의 유사도 점수를 계산한다. 사용되는 유사도 점수가 여러 개 일 경우 각각의 점수에 weight를 주어 최종 유사도 점수 합을 계산한다. 계산된 유사도 점수가 높을수록 의심스러운 스테이트먼트에 적용하기 적합한 후보 패치로 취급하기 위해 오름차순으로 정렬하여 우선순위화 한다.

3.3 데이터 셋

평가를 위해 사용된 데이터 셋은 표 1 과 같다. Defects4j 벤치마크내에 ConFix가 옳은 패치를 생성할 수 있었던 버그 22개 중 deprecated된 Closure-93 버그를 제외한 21개의 버그를 대상으로 패치 생성 및 우선순위화를 진행하였다.

3.4 실험 환경

패치 우선순위화에 대한 분석을 진행하기 위해 기존 패치 검증 과정 및 종료 과정을 일부 변경하였다. 후보 패치를 제한 시간 없이 제한 수(20,000)까지 혹은 생성할 수 있는 패치 재료가 떨어질 때까지 생성한 후 우선순위화를 거쳐 우선순위화한 순서대로 패치 검증을 진행한다. 패치 검증 과정 또한 시간 제한 없이 생성한 모든 후보 패치에 대해 검증을 거친다.

4. 분석 결과

표2는 전체적인 분석 결과를 보여주고 있다. 총 개수는 해당 버그에 대하여 생성한 후보 패치의 총 개수이며 Original은 패치 우선순위화를 사용하지

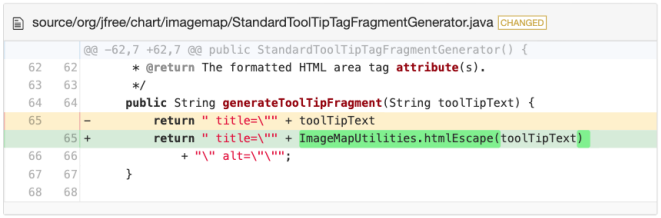

그림 1 Chart10 사람이 작성한 패치 예시

않았을 때 옳은 패치의 순위를 뜻한다. 각 표현 방식에 따른 우선순위화 결과는 각각 AST, 문자열, 벡터에 해당하는 열과 같으며 Average의 경우 각각 두 기법에 대한 유사도 점수의 평균을 우선순위화에 사용한 결과이다. 마지막으로 Total average의 경우 6개의 유사도 계산 기법의 점수 평균을 우선순위화에 사용한 결과이다.

패치 우선순위화가 옳은 패치를 더 높은 순위로 올릴 수 있었던 경우는 전체 21개 버그 중 14개였으며 나머지 7개에 대해서는 모든 유사도 기법들이 옳은 패치를 더 높은 순위로 끌어올릴 수 없었다.

표현 방식별 유사도 기법들의 평균 점수 사용 결과를 보았을 때 AST, 문자열, 벡터 각각 4개, 6개, 11개의 버그에 대해 옳은 패치의 순위를 높임에 따라 벡터가 가장 성능이 좋은 것을 확인하였다. 이는 기법 개별로 살펴보았을 때도 확인할 수 있는데, 코사인 유사도, 유클리디안 유사도를 사용했을 때 각각 10개, 12개의 버그에 대해 옳은 패치의 순위를 높일 수 있었다.

다만 Genealogy context 또한 21개의 버그 중 13개의 버그에 대해 옳은 패치의 순위를 높이는 데 성공했는데 이는 AST 노드 유형을 활용하여 패치 우선순위화를 했기 때문이다. 분석을 위해 구현에 사용된 벡터화 도구 Deckard의 알고리즘은 AST 노드 유형별 개수를 세어 벡터화를 진행한다. Genealogy context 또한 중복 노드 개수와는 다르게 유사도 계산에 AST 노드 유형 정보를 활용하였기 때문에 벡터 기반 유사도 기법들처럼 다수의 버그에 대해 옳은 패치를 높은 순위로 끌어올릴 수 있었다.

chart10를 포함한 7개의 버그에 대해서는 우선순위화를 통해 옳은 패치를 높은 순위로 올릴 수 없었다. 그중 5개의 버그들은 새로운 코드를 삽입하는 패치라는 공통점을 가지고 있었다. 그림 1은 Chart10의 사람이 작성한 패치의 예시를 보여주고 있다. APR 과정에서 생성된 옳은 패치 또한 그림 1과 동일한데 이 경우 htmlEscape이라는 메소드 호출을 새로 삽입하고 있다. 기존의 유사도 계산방식의 경우 의심스러운 스테이트먼트와 후보 패치를 적용한 스테이트먼트간의 유사도를 계산하기 위해 텍스트, 노드 유형 등의 일치하는 부분이 있어야 하지만 이 경우 새로운 코드가 삽입되었기 때문에 기존 유사도 기법으로는 옳은 패치를 높은 순위로 끌어올릴 수 없었다.

표 2 분석 결과

구분	Bug Id	총 개수	Original	AST			문자열			벡터			Total
				Overlap	Gen	Average	LCS	Context	Average	Cos	Eu	Average	Average
패치 우선순위화가 효과가 있는 그룹	Chart1	20,000	973	2,938	**138**	2433	**248**	**18**	**150**	**589**	**85**	**85**	**264**
	Chart11	8,002	3,069	4,286	**770**	4120	**77**	**1,980**	**478**	**567**	**567**	**567**	**503**
	Chart24	2,182	40	576	**9**	411	42	364	188	**9**	**9**	**9**	**35**
	Closure14	20,000	443	8,412	**143**	8243	4,194	8,629	6087	**354**	**124**	**124**	1471
	Closure38	20,000	15,127	**6,415**	**3,091**	**5,846**	**829**	**810**	**471**	**6,581**	**2,397**	**6,581**	**892**
	Closure73	20,000	17,175	**2,936**	**5,844**	**2614**	**3,960**	**1,139**	**2253**	**4,852**	**15,550**	**14741**	**4369**
	Closure92	20,000	683	8,851	**137**	8428	3,194	5,813	4070	3,124	**95**	3124	2325
	Closure109	20,000	969	14,431	**202**	14225	9,176	10,007	9891	**884**	**153**	**153**	7502
	Lang6	17,681	7,625	**5,171**	**1,677**	**4685**	**1,385**	**3,734**	**2325**	**123**	**1,064**	**857**	**763**
	Lang57	594	79	214	167	153	**15**	263	219	212	87	107	201
	Math5	3,830	304	3,104	**27**	2946	863	3,110	2854	**105**	**22**	**22**	2636
	Math33	20,000	19,620	**6,992**	**5,170**	**6364**	**2,822**	**9,723**	**5649**	**3,686**	**3,792**	**3791**	**1514**
	Math75	11,953	212	3,342	**79**	3007	664	2,760	1221	985	**62**	**62**	628
	Time19	20,000	25	476	**4**	256	3,248	2,704	2439	5,464	3,756	5363	2011
패치 우선순위화가 효과가 없는 그룹	Chart10	158	26	36	78	48	111	66	74	76	80	80	61
	Lang24	20,000	5,408	11,597	10,208	11991	13,247	10,045	11773	9,873	6,416	7946	11948
	Lang26	20,000	1,050	8,026	10,828	8760	3,942	2,838	3091	9,388	3,207	4058	5639
	Lang51	2,679	1,052	2,626	2,626	2626	2,626	2,626	2626	2,626	2,626	2626	2626
	Math30	20,000	545	11,225	10,289	11093	4,075	4,279	3994	10,178	4,889	8421	7529
	Math34	4,748	146	3,585	4,235	3754	3,887	3,555	3632	4,028	3,526	3979	3780
	Math70	5,623	173	1,404	2,514	1311	500	1,304	825	2,762	831	1140	1161

Overlap: 중복 노드 개수, **Gen**: Genealogy Context, **Context**: Contextual Similarity, **Cos**: 코사인 유사도, **Eu**: 유클리디안 유사도

5. 결론

본 논문에서는 유사도 계산에 사용되는 소스 코드의 표현방식에 따라 패치 우선순위화에 끼치는 영향을 확인하였다. 분석 결과 노드 유형 정보를 활용하는 유사도 계산 방식이 우선순위화에 효과가 있었고 그 중 소스 코드를 벡터화하는 방식이 효과가 있었다. 또한 새로운 코드를 추가하는 패치의 경우 기존의 패치 우선순위화 기법으로는 높은 순위로 끌어올리기 힘들었다. 향후 연구로 해당 표현 방식들에 대한 일반화와 코드 삽입 패치에 대해서 유사도를 측정할 수 있는 방식을 연구하고자 한다.

Acknowledge

이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단-차세대정보컴퓨팅기술개발사업 및 한국연구재단의 지원을 받아 수행된 연구임(No. 2017M3C4A706817923, No. 2019R1A2C200641112).

참고 문헌

[1] Moumita Asad et al., “Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair,” IEEE International Conference on Software Maintenance and Evolution, 2019.

[2] Ripon K. Saha et al., “Elixir: Effective Object-Oriented Program Repair,” IEEE/ACM International Conference on Automated Software Engineering, 2017.

[3] Xuan-Bach D. Le et al., “S3: Syntax-and Semantic-Guided Repair Synthesis via Programming by Examples,” Joint Meeting on Foundations of Software Engineering, 2017.

[4] Ming Wen et al., “Context-Aware Patch Generation for Better Automated Program Repair,” International Conference on Software Engineering, 2018.

[5] Zimin Chen et al., “The Remarkable Role of Similarity in Redundancy-based Program Repair,” arXiv:1811.05703, 2018.

[6] Jindae Kim et al., “Automatic Patch Generation with Context-based Change Application,” Empirical Software Engineering, 2019.

[7] Tao Ji et al., “Automated Program Repair by Using Similar Code containing fix ingredients,” Annual Computers, Software, and Applications Conference, 2020.

[8] Lingxiao Jiang et al. “Deckard: Scalable and accurate tree-based detection of code clones,” International Conference on Software Engineering, 2007.